A STUDY OF FAULT-TOLERANT COMPUTING

STANFORD RESEARCH INSTITUTE

PREPARED FOR
OFFICE OF NAVAL RESEARCH
ADVANCED RESEARCH PROJECTS AGENCY

JULY 1973

Final Report *31 July 1973*

A STUDY OF FAULT-TOLERANT COMPUTING

By: P. G. NEUMANN J. GOLDBERG
K. N. LEVITT J. H. WENSLEY

Prepared for:

DIRECTOR, INFORMATION SYSTEMS PROGRAM
MATHEMATICAL AND INFORMATION SCIENCES DIVISION
OFFICE OF THE NAVY
800 NORTH QUINCY STREET
ARLINGTON, VIRGINIA 22217

PROJECT MONITOR JOEL TRIMBLE

CONTRACT N00014-72-C-0254
ARPA Order No. 1998
Program Code No. 2P10

SRI Project 1693

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 U.S.A.

A STUDY OF FAULT-TOLERANT COMPUTING: FINAL REPORT

by
Peter G. Neumann
Jack Goldberg
Karl N. Levitt
John H. Wensley
Computer Science Group
Stanford Research Institute
Menlo Park, California
31 July, 1973

ARPA Order Number
1998, 27 December 1971

Program Code Number
2P10

Name of Contractor
Stanford Research Institute
Menlo Park, California 94025

Effective Date of Contract
12 January 1972

Contract Expiration Date
14 May 1973

Amount of Contract
$149,700.00

Contract Number
N00014-72-C-0254

Principal Investigator
Peter G. Neumann,
Phone 415-326-6200,
ext. 2375

Scientific Officer
Director, Information
Systems Program
Mathematical and Information
Sciences Division
Office of the Navy
800 North Quincy Street
Arlington, Virginia 22217

Short Title of Work
FAULT-TOLERANT COMPUTING

Sponsored by and prepared for the
Defense Advanced Research Projects Agency
ARPA Order Number 1998

(Form Approved Budget Bureau No. 22-R0293)

Approved:

David R. Brown
David R. Brown, Director,
Information Science Laboratory

Peter G. Neumann
Peter G. Neumann,
Principal Investigator

SRI Project 1693

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author) Stanford Research Institute	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3 REPORT TITLE

A STUDY OF FAULT-TOLERANT COMPUTING: FINAL REPORT

4 DESCRIPTIVE NOTES (Type of report and inclusive dates)

Final report covering 12 January 1972 - 15 May 1973

5 AUTHOR(S) (First name, middle initial, last name)

Peter G. Neumann, Jack Goldberg, Karl N. Levitt, John H. Wensley

6 REPORT DATE 31 July 1973	7a. TOTAL NO OF PAGES 236	7b. NO OF REFS 61
8a. CONTRACT OR GRANT NO N 000 14-72-C-0254 (ONR) b. PROJECT NO c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	

10 DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited

11 SUPPLEMENTARY NOTES None	12 SPONSORING MILITARY ACTIVITY Defense Advanced Research Projects Agency

13 ABSTRACT

This report presents the results of a study of fault-tolerant computing. Existing and new architectural techniques are evaluated for use in cost-effective systems attaining desired measures of correctness, availability and recovery. Various architectures and applications are considered. Appendices contain a brief census of 35 fault-tolerant systems, and a concise survey of 17 representative systems, as well as detailed results on reliable memories and arithmetic.

DD FORM 1 NOV 65 1473 (PAGE 1) PLATE NO. 21856

S/N 0102-014-6600

Unclassified
Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Fault-tolerant computing						
Computer reliability						
Computer availability						
Architecture						
Computer systems						

PREFACE

This report has been prepared with the aid of an on-line graphical text editor deficient in its lack of underlining. UPPER CASE is used throughout in place of underlining. We hope that this causes no confusion.

References in the text are cited by author name(s) and year, and are given in Chapter 8. References cited with an "A2" instead of the year refer to system descriptions contained in Appendix 2. For example, "(Wensley 72)" refers to a reference provided in Chapter 8, while "(Wensley A2)" refers to the description of a particular system found in Appendix 2. Further references are also found in each of the appendices.

In the light of the existence of several extremely comprehensive bibliographies in the area of fault tolerant computing (cited at the beginning of Chapter 8), we have chosen to be selective in our references. Where a multiplicity of references is relevant, we have sometimes chosen to cite only the most recent ones, so that the interested reader can pursue earlier references by indirection.

CONTENTS

ILLUSTRATIONS

TABLES

A STUDY OF FAULT-TOLERANT COMPUTING: FINAL TECHNICAL REPORT

Peter G. Neumann, Jack Goldberg, Karl N. Levitt and John H. Wensley
Computer Science Group, Stanford Research Institute, Menlo Park, CA

CHAPTER 1. SUMMARY OF THIS REPORT

This report presents the results of a study of the state of the art of designing fault-tolerant computing systems. This chapter provides a summary of the technical problem, the technical results, the relevance of the study to the Department of Defense, to users and to vendors, and implications for future research and development.

1.1. THE TECHNICAL PROBLEM

The purpose of this study is

* To survey and evaluate existing systems, system concepts, and relevant existing theory, in order to assess the art of designing effective and economical fault-tolerant systems.

* To define and evaluate new approaches to the design of computing systems with improved fault tolerance.

The system goals of interest include the attainment of:

* CORRECTNESS-- High degrees of correct operation despite the occurrence of faults in hardware.

* AVAILABILITY-- Very high system availability (i.e., very little down-time), with little or no emergency maintenance and possibly very little maintenance at all.

* RECOVERY-- Rapid recovery from faults not immediately tolerated, with limited but known (and usually recoverable) losses.

* ECONOMY-- Low redundancy relative to system replication, and low cost of fault tolerance relative to the total environment in which the computer system exists.

System goals are considered that might require massive equipment redundancy (e.g., for extremely high correctness, or very long zero-maintenance lifetime, or extremely fast recovery), but these goals are not of primary interest here.

1.2. TECHNICAL RESULTS

The basic conclusion of this study is that substantial fault tolerance can be achieved at surprisingly low cost (in both hardware and software) under a wide range of operating requirements. The fault tolerance attainable with the present state of the art is much greater than in present systems, and satisfies many present demands. Further improvements are also possible that would allow design of still more powerful systems. This basic conclusion is especially applicable to large systems with flexible real-time requirements. Such systems include general-purpose systems, message store-and-forward systems, communications processors, and networks.

In Chapters 3, 4, and 5 of this report, we review many techniques for fault tolerance. These techniques facilitate the detection, isolation, location, and removal of errors, and the recovery from the effects of these errors. In Section 3.3, system architectures are considered, including a variety of simplex and multiprocessor configurations. In Chapter 6, the techniques for fault tolerance are applied to these architectures. Quantitative measures of system correctness, availability, recovery, and cost are given for each of these architectures.

We find no fundamental gaps in the state of the hardware design art preventing the attainment of high degrees of fault tolerance at low cost relative to the overall system -- except for questions of recovery speed and very long unattended life discussed below. On the one hand, there

are applications in which the computer systems represent only a small portion of the total costs, e.g., in special-purpose control applications. In these cases, the extreme solution of replication of computer equipment with comparison or voting may be economical overall. In most existing systems and system designs for such applications, which provide guaranteed fault tolerance for essentially all single faults, anywhere from 60 to 80 percent of the hardware is typically devoted to fault tolerance. On the other hand, we find in general that 10 to 40 percent of the hardware devoted to fault tolerance is sufficient to achieve adequate correctness and availability for many systems, except when all system results have highly critical real-time requirements on correct performance. Such low redundancy can be achieved by a combination of techniques (both existing and newly developing), and by careful use of structure in the system. Such structure facilitates taking advantage of the nonuniformity of internal system requirements, and permits various fault-tolerance techniques to be used when and where they are most effective, rather than uniformly. The resulting partitioning makes complete single-fault tolerance necessary only within certain critical partitions. It also facilitates speedy recovery when essential. Such structure also facilitates graceful degradation of performance.

We do find fundamental gaps in the art of designing and implementing software to support hardware facilities for fault tolerance. This art is notably weak in the areas of specifying and verifying system designs and implementations in a way that unifies hardware and software with proper consideration of operational needs. This weakness is especially evident with respect to the poor state of operating systems, and in the adequate coverage and resolution of system diagnostics. As noted above, present systems and the present design art are seriously deficient in the speed of recovery following faults. Solutions to this problem require advances in hardware and software for improved diagnosability, and in total hardware- software integration. The art is also weak in maintaining smoothly degradable performance in a low-maintenance environment.

In modern computer design, boundaries between hardware and software are becoming increasingly diffuse. There is a real need to upgrade the fault-tolerance design art so that it can become a standard facet of computer design. Significant effort is required in system software and system operations to assure that good hardware development is not compromised. We emphasize the critical importance of developing balanced system designs well suited to particular system needs. It is helpful if system goals can be integrated from the beginning, rather than retrofitting fault tolerance into a system not designed with it in mind. To this end, the concept of system structure is useful in all the stages of system development, including the design, implementation, verification, use, operation, and evolution of the system. Structured design and implementation hold great promise for improving the art, not just for fault tolerance, but for computer system design in general.

In many cases, high availability cannot be achieved without a secure system (employing protection mechanisms in the hardware and operating system to assure that it is relatively crash-proof). In turn, such security cannot be achieved without high reliability, especially in the portion of the system that affects security. System structure can also be helpful in achieving the goals of security.

Our conclusions also include implications of system design on the operational and human aspects, which play a critical role in keeping a system highly available. These include not only considerations affecting correctness, continued system availability, and rapid recovery, but also those lessening the critical dependence on administrators, skilled operators, and maintenance personnel.

We conclude that the attainment of a sufficiently fault-tolerant system is possible for various particular applications at relatively low cost. However, considerable care and common sense are still required in system implementation. Our survey of existing systems shows that seemingly obvious measures for fault avoidance are often ignored. If the reader occasionally finds a statement that seems obvious, it may be included here for completeness, or because it serves as a basis for subsequent

discussion, or because there are hidden difficulties in implementation. We pose the challenge to practitioners and theoreticians of fault tolerance to find structures and theories that move these "obvious" design decisions from the domain of good judgment to that of systematic practice.

Many of the techniques discussed in this report are useful with present-day technologies. Others are particularly suited to emerging technologies such as LSI, which can have a significant effect on system fault tolerance, e.g., due to compactness, low heat generation, and low cost. These latter technologies will permit the use of techniques not previously practical. However, the trend to high-density systems using advanced technologies (for memory as well as processing) will not obviate the need for architectural measures to achieve fault tolerance. It is true that the unit reliability of new LSI devices is not much less than IC and MSI devices, while the devices are significantly more powerful. Thus, a given function may be realized in LSI with higher reliability due to the use of fewer devices. However, while the number of devices per function decreases, there is a strong tendency for large general-purpose systems to grow, up to the limits of cost. For example, there is a trend toward increasingly powerful hardware in order to simplify programming.

An additional factor is the limit on reliability imposed by the high cost of testing a device to an assured level of reliability. While a device may be extremely reliable, the system designer can assume only that reliability that can be demonstrated. The current practical limit on testable failure rates for a device ranges from 10^{-6} to 10^{-7} failures per hour. This implies that, for very large systems, the projected system failure rate would be 10^{-1} to 10^{-2} failures per hour. This clearly requires system-level fault-tolerance measures.

The work reported here is novel in several respects. It represents both theoretical and practical approaches to economical fault tolerance, rather than the use of massive redundancy. It provides a framework for a unified hardware/software approach to system design for fault

tolerance. It attempts to show explicit cost figures for fault tolerance over a wide range of architectures. It also includes several new theroretical results on reconfigurable memories and on coding for arithmetic.

1.3. RELEVANCE OF THIS STUDY

This work is applicable to many kinds of computing systems. These include systems with general-purpose and/or special-purpose capability, and network control computers such as the interface message processors (IMPs) in the ARPA network.

1.3.1. RELEVANCE TO THE DEPARTMENT OF DEFENSE

Specific conclusions of our study affecting the Department of Defense include the following.

* Significantly better fault tolerance (e.g., correct behavior, high availability, rapid recovery, and high system security) can be obtained, even in the presence of malfunctions.

* Significantly more economical fault tolerance can be achieved, with more efficient use of redundancy, more remote diagnosis and maintenance, more automatic self-maintenance (e.g., the use of spares, with automatic reconfiguration), and less emergency maintenance. More automatic operation will result in reducing the unnecessary reliance on potentially unreliable people in critical positions.

* While the primary scope of this report involves the design of large general-purpose systems, there is considerable potential for applicability to tactical and other real-time control systems.

* Significant effort must be expended to assure overall system reliability, e.g., effort concerning good software design and implementation, reliable operations personnel, and other system support.

Otherwise, good hardware design may be wasted. In addition, history shows that computer manufacturers have been slow to respond to customer needs that have not been clearly and forcefully enunciated. We feel that if DOD wishes to have systems with economical fault tolerance, it must stimulate manufacturers to develop such systems by defining and enforcing fault-tolerance requirements in terms of realizable specifications.

1.3.2. RELEVANCE TO USER COMMUNITIES

The recommendations here generally make most of the mechanisms for achieving high reliability and high availability invisible to system users during system operation. However, user communities will have to exhibit greater awareness of what can be achieved and what they might require. They should clearly define their needs, and exhibit considerable unity in presenting these needs to the vendors.

1.3.3. RELEVANCE TO THE VENDORS

In recent years several commercial vendors have undertaken serious effort toward achieving fault tolerance in computer systems, primarily in the light f aerospace needs. Several useful steps forward have also been taken in some recent commercial systems, such as the use of error-correcting codes and instruction retry, and the use of hierarchical recovery strategies. It is hoped that this report will be helpful to all system development efforts in focusing attention on fault tolerance as an integral part of system development, especially since much can be done at low cost. Some of the techniques herein can be retrofitted onto existing systems. However, it is most cost-effective to integrate fault tolerance into the overall design.

1.4. IMPLICATIONS FOR FUTURE RESEARCH AND DEVELOPMENT

While our basic finding is that the present art provides the basis for reliable systems at reasonable costs, there exist limitations which, if overcome, could result in further significant improvements (e.g., by

reducing recovery time, by reducing the residual error rate, and by further reducing the cost). In Chapter 7, we summarize some recommendations for achieving such improvements. These include better techniques for error detection and fault diagnosis, novel architectures specifically suited to fault tolerance, and significantly improved techniques for the analysis of fault-tolerant systems.

CHAPTER 2. INTRODUCTION

As used here, the term "fault tolerance" is used broadly to mean the ability of a system to withstand various kinds of hardware malfunctions and mishaps. There are varying degrees of fault tolerance, including continued correct performance for some portion of the system, and continued availability of some portion of the system, although possibly with degraded capacity. There are increasingly many applications requiring much better fault tolerance than is currently available. Those of interest here include general-purpose systems with both batch and interactive capabilities, as well as various special-purpose systems such as message switching systems. Our emphasis is on economical fault tolerance for applications with varying real-time criticalities. The work is also relevant to various aerospace applications that are currently approached with massive redundancy.

We are concerned primarily with system-level techniques for increasing fault tolerance, rather than with techniques for improving the reliability of various technologies. Thus, we focus largely on system architecture. This chapter provides an introduction to the report. Chapter 3 gives a guide to the techniques for fault tolerance useful at various system levels, and illustrates their applicability to system and network architecture. Included are simplex systems and multiprocessors (with widely varying degrees of parallelism, independence, and common information access). The chapter also discusses the role of structure in the attainment of economical fault tolerance. Chapters 4 and 5 present some advances in architectural techniques for fault tolerance, Chapter 4 considering memory, and Chapter 5 considering arithmetic, logic, and control. Chapter 6 considers different application fields (special-purpose, aerospace, communications, etc.) and presents the special requirements for fault tolerance in each field. From these special requirements, appropriate techniques and architectures are derived, and their effectiveness considered. Chapter 7 provides the conclusions of our study, along with specific recommendations for future research.

Several appendices are included. Appendix 1 provides a census of fault tolerant systems. Appendix 2 provides a detailed survey of various representative systems. Appendix 3 gives substantially greater detail to support the memory organizations of Chapter 4. Finally, Appendix 4 presents some new results on byte coding for arithmetic.

2.1. BASIC DEFINITIONS AND ASSUMPTIONS

In this section we present definitions of the basic terms associated with fault-tolerant systems. In addition, we present a few assumptions that have guided us in the design approaches considered here.

FAULTS, ERRORS AND FAILURES

The terms "fault" and "error" are defined with respect to the interface of a hardware or software mechanism, e.g., a component or a subsystem, whose output is observable at least to some other mechanism. An ERROR is a disparity between the actual output at such an interface and the value expected under normal operation. Examples are an incorrect result from an arithmetic unit, an incorrect word in a memory unit, and an incorrect word involving an input-output device. Errors may be SINGLE or MULTIPLE, depending on their nature. For example, an additive error in a single bit position of an adder could affect several bit positions (with carries), and would appear to be a multiple error in memory. Errors may be DETECTED or UNDETECTED at a particular interface. For example, single memory errors are detected by simple parity checking in memory, but double errors (or quadruple errors, etc.) are not. Errors not detected at one interface may be subsequently detected at another (higher-level) interface, e.g., via consistency checks.

A FAULT is an internal malfunction within a mechanism. It may or may not result in an observable error. This depends on the data that are actually entered to the mechanism, whether or not the faulty part is redundant, and whether or not the mecanism has internal fault-tolerance capability. Faults may be transient, intermittent, or permanent. A

TRANSIENT fault is one that occurs once, leaving the hardware in a fault-free condition, but with possible effects on the software and on system operation. An INTERMITTENT fault is one that recurs, with intervening fault-free periods. A PERMANENT fault is one that persists steadily without interruption. In hardware a transient fault may become intermittent, and an intermittent fault may become permanent. (The terms "transient" and "intermittent" are often merged.) A transient fault might be caused, for example, by interference on a bus. A permanent fault might be due to a shorted transistor, shorted wires, an open connection, or a power supply fluctuation, for example.

Faults are hardware phenomena, and are potential sources of system errors. Other sources of system errors also exist, e.g., mistakes in design, or misuse. Examples of potential sources of errors are found in Table 2.1. (See also Yourdon 72.)

The mechanisms of transient faults are not so well understood as, for example, permanent faults, but several observations are relevant here.

* In many technologies, transient and intermittent faults seem to dominate permanent faults by at least an order of magnitude. This dominance is partly because the nonpermanent faults are harder to find, and thus are usually not found before they can recur. If they degenerate to permanent faults, they usually become more readily identifiable.

* A major cause of errors is poor design, e.g., in not properly handling the occurrences of exceptional cases (e.g., electrical disturbances). Examples of such cases are undesirable circuit coupling that is data dependent, unusual timing dependencies, and marginally designed power supplies.

INDEPENDENCE. An important property of multiple faults and multiple errors is their relative INDEPENDENCE or DEPENDENCE. In the case of an LSI realization, a fault within a chip can result in multiple (dependent) errors from that chip. Faults in different chips should be considered as independent if adequate protection exists at chip

interfaces. For conventional core memories, core and line driver faults seem to occur independently of one another. Thus, for each subsystem there is a primitive element or a set of primitive elements to which faults can be ascribed.

The events following a fault are summarized in Figure 2.1. When a fault is detected (e.g., via coding or duplication, or implicitly by fault masking), a recovery strategy is invoked. However, as long as a fault remains undetected, the effects of the fault may propagate. It may even be compounded by further (dependent or independent) faults or by being a REPEATED-USE fault (Avizienis 72). In many cases faulty behavior is ultimately detected (although in extreme cases perhaps only by complaints following a system crash), at which point recovery is attempted.

The possible effects of undetected errors are quite varied. There is a wide range of effects of faults on system behavior. There are many forms of "crashes", gradual or sudden, impairing in varying degrees correctness, availability, performance and security. However, it is not necessary that all errors be detected in all situations. For example, in a time-sharing environment most users are willing to accept occasional errors due to hardware faults, provided either they or the system can detect the errors, and provided adequate recovery and file integrity are available. Users are normally not willing to accept frequent crashes, long outages, or loss of on-line files maintained by a system whose intent is to eliminate the need for private backup. In usage here, a FAILURE is an error whose effect is in some sense critical. Various senses of "critical" are discussed in Section 3.2.2..

RELIABILITY, CORRECTNESS, AVAILABILITY, AND FAULT TOLERANCE.

FAULT TOLERANCE is (roughly speaking) the ability of a system to withstand faults. The significant effects under consideration here are LOSS OF CORRECTNESS (e.g., as the result of errors in processing and in storage -- the latter including damage to stored programs) and LOSS OF AVAILABILITY (e.g., the loss of computing capacity or storage capacity,

Table 2.1
SOME SOURCES OF SYSTEM ERRORS

SOURCES OF HARDWARE FAULTS Physical bonds and loose connectors Wear in moving parts Material aging Insulation breakdown Environmental effects (e.g., temperature, humidity, vibrations, electrical and electromagnetic disturbances) Human-induced breakage
OTHER SOURCES (HARD, SOFT, OPERATIONAL) Inadequate design and implementation: Lack of checking and validation in interfaces, especially in response to unanticipated conditions Sensitivity to timing variations Data dependency effects Usage-induced hardware damage Inadequate system security Inadequate system verification Acts of God (lightning, floods, etc.) People problems (e.g., administration, maintenance, concurrent development, operators, documentation) Power sources, local and public utilities Support functions (e.g., air conditioning)

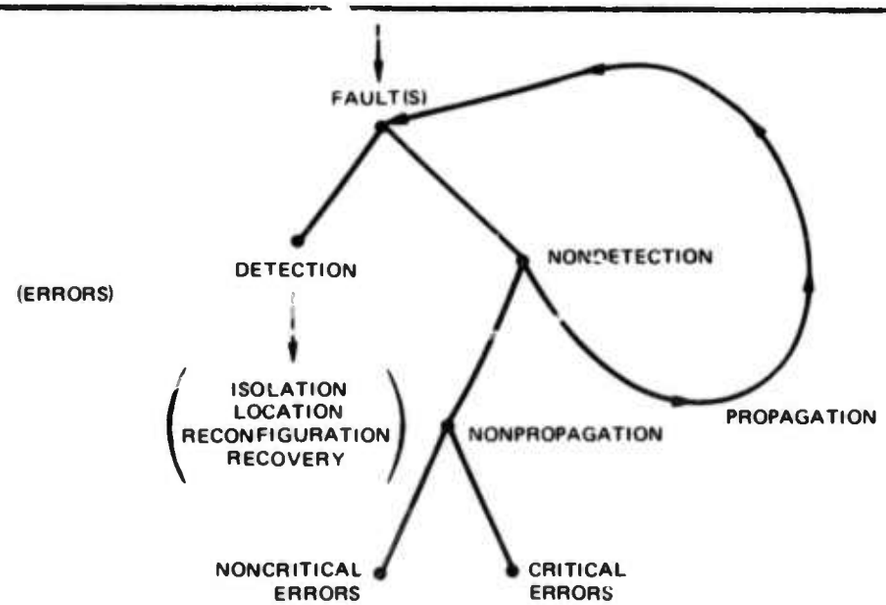


FIGURE 2.1 MODEL OF FAULTY BEHAVIOR

or of response time). A violation of system security may lead to loss of both correctness and availability, as well as loss of other aspects. The absence of an expected output may also lead to the loss of availability or correctness or both, depending on the application. AVAILABILITY is thus a measure of having operative resources that can be called upon to handle a task. CORRECTNESS is a measure of how error-free a result is at some interface of interest. The term "RELIABILITY" is little used in this report in its standard meaning of the probability of correct behavior at a specific time. The term "RELIABLE" is used in a qualitative sense to denote correctness and/or availability.

As a means for evaluating a given system, it is desirable to derive quantitative measures of correctness and availability. The classical measures "mean time to failure" and "mean time to repair" are not by themselves adequate measures for most complex systems. Better measures are probabilities as a function of time that certain resources are available and that certain data are correct.

It is readily seen that a wide range of effects is possible. For a given fault (or combination of faults), these effects may range widely in their fault tolerance between two extremes -- from complete fault tolerance (with no incorrect results visible externally) to a total collapse of the system. In the latter extreme there may be extensive loss of correctness and availability for a protracted time during and after the collapse, and lengthy delays until correct performance and adequate capacity are again available. Between these extremes are various forms of partial collapse, with varying degrees of recoverability. The early detection of faults is also important to prevent potential security violations that may result from faulty behavior. Upon detection, diagnostic procedures can be used to assess the scope of the error propagation, and appropriate recovery procedures initiated.

There are two ways of using the concepts of availability and correctness to design a system. First, for many applications one design goal is to

eliminate down-time entirely or at least to reduce it to a negligible amount. "Negligible" might mean seconds in the case of a telephone utility, or minutes in the case of a time-shared facility, but in any event the intent is to keep the machine running despite faults, pending maintenance. For such applications it is usually sufficient to provide single-fault tolerance for such critical functions as file handlers, memory managers, and restart and recovery procedures, plus sufficient redundant hardware so that a working system can be configured after the occurrence of each fault. Second, for applications where the computer is so remote as to preclude maintenance, the important issues are: (a) the probability that the computer has sufficient resources left after a period of time, and (b) the probability that correct answers are produced for certain critical functions. The aerospace environment is perhaps the main current example of this approach, although some transportation systems, electric power systems, financial systems and secure systems have also been beneficiaries of fault-tolerance techniques. In any event it might be necessary for a computer in such an environment to tolerate many faults.

REDUNDANCY. An important measure of the effectiveness of any fault-tolerant system is its REDUNDANCY. Let "k" be the cost of hardware needed in the absence of any fault-tolerant requirement, and let "r" be the cost of extra hardware needed to achieve fault-tolerant behavior. Then the relative redundancy "R" is $R = r/(k+r) = r/n$, as a fraction of the total cost "n" of the system. (This measure is used more or less exclusively throughout this report, rather than the alternative approach of citing the percentage increase over a comparable intolerant machine, e.g., 200 percent for triplication.) This definition is consistent with the coding theory concept of redundancy, in which k, r, and n are measured in bits. Except in relatively trivial system configurations, it is a difficult chore to estimate the redundancy. For a system that just employs triplication of certain hardware blocks together with appropriate voters, the redundancy is $(2+v)/(3+v)$, where "v" is the cost of the hardware voters relative to the functional block. The evaluation of redundancy is more difficult in a situation where, for example, a multiprocessor is used solely to

achieve fault tolerance. That is, if fault tolerance were not a requirement, then a conventional uniprocessor might suffice. Here "r" must include numerous items, for example,

* Storage area to hold reconfiguration programs
* Extra processing power to overcome the multiprocessor penalty
* Redundant busses
* Cache memories to overcome bus traffic delays
* Switches to accomplish reconfigurations
* Storage areas to hold rollback status.

Similar measures are meaningful for software costs and execution time.

The priorities among the system goals may have major effects on the resulting systems, and may call for widely differing architectures. In our consideration of various architectures in this report, we attempt to evaluate the redundancy required, at least with sufficient accuracy that a gross estimate of system cost is possible. Included are both low-redundancy design approaches for single-fault tolerance and higher-redundancy approaches for multiple-fault tolerance.

2.2. SEVERAL ILLUSTRATIONS OF FAULTY SYSTEM BEHAVIOR

Several recent examples of failures in contemporary systems are instructive. The first provides a perspective overview, and concerns Multics (Saltzer A2), a system with little hardware redundancy but with file availability attained through software. Here outages fall into three roughly equal categories: hardware, software, and operations. To make matters more complicated, system development typically has gone on concurrently with the operation of the production system, either simultaneously on the same (two-processor) system or separately with the two processors partitioned. The hardware problems are fairly traditional (e.g., processor problems, memory errors, etc.), although the Multics software is tolerant of many input-output and secondary storage errors in terms of providing continued availability. The software problems are due mostly to new bugs introduced by the concurrence of the development effort, with new system versions being

installed as often as once a week. (This is in contrast to OS/360, in which it appears that even the occasional new "debugged" release had some large number of bugs -- the constant "1000" is popularly cited.) Similarly in operations, at least half of the problems are the direct result of the development process, arising through manual reconfiguration (due to a hardware design not intended for dynamic self-reconfiguration), or through changes in operating procedure. The remaining operational problems are typical, e.g., power outages. Thus, about half of the problems are attributable to the coexistence of the development effort. The pattern of roughly equal distribution of failures due to hardware, software, and operations is found in many systems. The frequency of failures seems to diminish greatly if experimentation slows down and production is stressed.

The second example involves the outage of a No. 1 ESS (Ulrich A2) office in Nashville (at night), involving total outage of a few hours, with partial outage for ten hours. This was preceded by accumulated errors in the call store combined with inadequate responses of the operating and maintenance staff, eventually triggered by malfunctions in both halves of the system.

The third example concerns the Market Data System MDS1 of the New York Stock Exchange, operating with dual systems. After satisfactory system validation prior to the opening for business on Feb. 24, 1972, system A experienced a crash four minutes into the market session. Automatic recovery was successfully invoked within a few seconds by switching to system B, and correct operation continued with no loss. After off-line maintenance of system A, the contents of drum B were copied onto drum A, and both drums were again on-line. Unfortunately, during the time since the morning validation, drum B had developed a faulty master record, which was subsequently accessed. This caused system B to halt. Control was automatically switched to system A, whereupon system A also halted on the copy of that record. Manual recovery was lengthy, and the total outage lasted 29 minutes, the worst in seven years of operation. (The New York Stock Exchange has since cut over to MDS2, with three 360/50s and a duplicated large core storage for files.)

The fourth example is that of a telephone system in Kuala Lumpur which collapsed twice in two years, with significant hardware damage. Subsequent analysis finally determined that each of these events occurred just a few minutes before post time on the day of the annual horse race, damaging part of an exchange serving a community noted for its gambling spirit. Unfortunately, the operations personnel were all at the track at the time, and could not notice the sudden overload in attempted calls.

The first example illustrates problems that can be overcome by administrative control and by further isolating a development effort from production. It also illustrates the enormous difficulty of discriminating among hardware-induced and software-induced errors. Multics and CP-67 provide environments in which noncritical development of software can be debugged on-line within a production system. Nevertheless, final debugging of critical system software is not easy without a separate system, including real users in a real environment. This problem is often very difficult, as in the case of the development for the Interface Message Processors (IMPs) in the ARPA Network.

The second example illustrates the typical overdependence on the need for good field engineers. In some cases high quality maintenance is possible (e.g., in the FAA air traffic control system, where the number of centers is small). In the ESS case, where many systems are involved, the problem becomes critical. If a skilled engineer is required at all times at each installation, the system is poorly engineered. If he is required only rarely, but then urgently, it is difficult to staff all centers with sufficiently skilled and motivated personnel. The need for systems not requiring emergency maintenance is thus very great, especially when many systems exist at distributed locations, each with strict availability requirements.

The third example illustrates the fact that software is usually never debugged and never finished, as demonstrated by an unanticipated situation which had never arisen in seven years' operation. There are

many other tales of systems in which long-standing hardware and/or software bugs (in this case the lack of validation during copying) were discovered only after years of operation. In some of these cases, considerable reexamination of the correctness of earlier results was required. Nevertheless, the MDSI system was quite remarkable in that it used off-the-shelf equipment and recorded a highly successful record of availability in its lifetime.

The fourth example illustrates the danger in taking advantage of an apparently reasonable design assumption. In this case it was clearly unwise to assume that traffic consisted of essentially independent random calls.

Chapter 3 which follows discusses techniques for fault tolerance, the effects of faulty behavior and the recovery from it.

CHAPTER 3. TECHNIQUES FOR FAULT TOLERANCE

This chapter reviews basic principles and techniques in the present art of design for fault tolerance, and demonstrates their use in realizing economical system architectures. Section 3.1 reviews existing and proposed design techniques for fault tolerance (applicable in hardware and in software). These include techniques for error detection, error confinement, fault location, reconfiguration, and recovery.

Section 3.2 examines the developing art of applying structure to system design and implementation, including the role of explicit structural levels in partitioning the hardware, the software, and the microware. Concepts of criticality are discussed. The use of time-space tradeoffs useful in facilitating economical fault tolerance is investigated. Also considered is the role of system structure in achieving rapid recovery from faults not completely tolerated.

Section 3.3. examines the application of these techniques to the realization of economical systems and networks. Various architectural types are considered. Their relevance to specific applications is discussed in Chapter 6.

Detailed techniques for memory, and for arithmetic and logic, are discussed in Chapters 4 and 5, respectively.

We assume here the use of intrinsically reliable technologies and of sound engineering practice (e.g., good component engineering and careful quality control). We recommend, but do not discuss in detail, the use of techniques for system modeling, reliability analysis, and the formal verification of design properties. We assume the existence of good system development practice (including the use of suitable development tools, e.g., languages, debuggers, and test environments) and good operating practice (e.g., avoidance of simultaneous system development except under highly controlled circumstances). These techniques are particularly important in the attainment of good fault tolerance.

We also recommend, but do not consider in detail, techniques for the achievement of a stable physical operating environment. These include the use of highly distributed reliable power supplies to minimize outage. For continuous availability, the use of standby batteries and generators is desirable.

3.1 DESIGN TECHNIQUES FOR FAULT-TOLERANT SYSTEMS

In this section, we classify and evaluate techniques for designing fault-tolerant computer systems. The basic techniques are summarized in Table 3.1. These techniques include novel techniques discussed in detail here and well-known techniques which are given for completeness.

The following operations are basic to the attainment of fault tolerance.

ERROR DETECTION. An error is detected when a discrepancy signal is received by some subsystem that can take action to circumvent the error.

ERROR CONFINEMENT. Errors should be confined as much as possible within particular interfaces until some correction mechanisms can be invoked.

FAULT LOCATION. A fault (faults) must be pinpointed to some unit.

RECONFIGURATION. A faulty unit must be removed, replaced, or worked around.

RECOVERY. In the case of error propagation, it may be necessary to restart some processes at some error-free state in order to perform lost computation and restore lost files. It may also be necessary to restore the system itself to a viable state.

Most of the techniques discussed here are fairly well known and well understood. We give special attention to some of the cases where more research is required. It is clear that a system can be designed to

tolerate faults occurring independently. The challenge is to achieve a design that is not too costly in terms of hardware, and that can tolerate realistic faults -- including certain dependent faults. However, the design should also be modifiable as reliability and availability needs change,

SELECTIVE AND DYNAMIC USAGE OF FAULT-TOLERANCE TECHNIQUES. The techniques of Table 3.1 may be used in different ways with respect to space and time. In space (e.g., within a memory or a processor), a technique may be used UNIFORMLY (one approach throughout) or SELECTIVELY (applied only in certain places). In time, it may be used STATICALLY or DYNAMICALLY. STATIC usage concerns actions with no changes over time in the operating environment or in the flow of control (e.g., fault-masking via coding, fixed replication with voting). DYNAMIC usage concerns fundamental variations in the control (e.g., in the sequencing or in the configuration), such as in detection followed by diagnosis, rollback, and replacement, or as in the use of replication configured only on demand of the software. As seen below, there are significant advantages (e.g., cost savings) that result from selective and dynamic usage. Examples of these modes of usage are given in Section 3.2.

Numerous specific aspects of each of these five operations are discussed below. For the present discussion, however, we wish to emphasize the following points:

* In relatively trivial fault-tolerant systems, not all of the five operations are distinctly identifiable. In a system that employs just triplication with voting, for example, the error confinement process embodies the other four operations.

* These operations can be carried out by varying combinations of hardware, software, and microcode. Fault tolerance, therefore, is a distributed function which may be implemented at various computational levels.

Table 3.1.
SUMMARY OF MAJOR DESIGN TECHNIQUES FOR FAULT TOLERANCE

DETECTION

Coding: error detection
Double-rail encoding for logic
Duplication in space or time and comparison (hard or soft),
Consistency checks (e.g., algorithmic checks, read after write, back substitution, partial Floyd assertions)
Probabilistic detection
Deferred detection
Detection as a byproduct of diagnosis, periodic or otherwise

PREVENTION OF ERROR PROPAGATION, AND LOCAL CORRECTION

Delaying the results until validated
Coding: error correction (Hamming, burst, byte)
Replication with voting
Isolation, e.g., via powering off, reconfiguration and fail-safe switching, fail-safe structural design (esp. involving protection and interrupts), use of read-only memories, asynchronous decoupling, clock independence

LOCATION OF FAULTS OR ERRORS

Coding: error location (also implicit in error correction)
Triplication (implicitly error locating)
Diagnosis, possibly with reconfiguration for testing

RECONFIGURATION AROUND FAULTY UNITS

Removal (deconfiguration) with degradation
Reconfiguration around a fault contextually (without its elimination)
Replacement by switching of standby spares
Replacement physically

RECOVERY

Single-instruction retry with buffered operands
Rollback to a program checkpoint, with manual or automatic checkpointing
Audit trails to facilitate subsequent recovery
Interpretive recovery (e.g., unwinding, salvaging, selective file retrieval)
Bootstrap recovery from fixed point (with side effects)

* In effectively using the fault-tolerance techniques selectively and dynamically, there are fairly well-defined trade-offs among the time required to carry out one of these operations, the hardware redundancy required, and the probability of successfully carrying out the operation. For example, deferred detection and/or correction of arbitrary logic may produce significant cost savings.

3.1.1. ERROR DETECTION

One of the main problems in achieving low-cost fault tolerance is the problem of achieving economical error detection. Aside from well-structured situations such as core memories, parallel adders, tape memories and bus transfers, error detection with less than 50% redundancy (duplication) has remained unsolved.

When a system fails, its failure is often obvious to a human. A terminal may appear dead (e.g., because of a system crash or a loop in his program), or his results may appear to be wrong. Internally, many harmful errors are similarly vast, e.g., involving alteration in the flow of control. The reasons for wide discrepancies between expected results and actual results include the following:

* A faulty logic circuit is sometimes used repeatedly in the absence of internal error detection, thus increasing the chance of a readily discernible error.

* Many simple hardware faults (permanent or transient) have a drastic affect on program control, e.g., directing control to an incorrect instruction or addressing the wrong memory location. Other faults may not affect control. Also, many computations do not allow simple consistency checks. Thus more general and problem-independent error detection mechanisms are essential.

In spite of the ease of some error detection to a human, error detection can be a costly operation when carried out automatically. An arbitrarily structured processor using known error detection techniques

seems to require at least 50% redundancy to permit immediate detection of all possible processor errors. Fortunately, this cost does not carry over to total system cost for error detection, for various reasons. For example, the processor may be a small part of the total system; much more palatable detection schemes exist for memories, channels, etc. Similarly, there is no need to detect all possible errors, and also immediate error detection is not needed. The basic methods are outlined below, leaving to later sections the detailed discussions and evaluations.

It is clear that errors can be detected with any desired degree of completeness, and with any desired degree of immediacy. The challenge is to achieve such detection with low redundancy.

3.1.1.1 ERROR-DETECTING CODES

Error-detecting codes make it possible to detect the first occurrence of an error at some particular interface, e.g., memory, channels, an arithmetic logic unit, a processor, or the entire computer. Section 3.1.1.3 below discusses the possibilities of deferred error detection.

Codes with a single parity bit for each memory word are widely used for error detection in memory. Such codes, with negligible redundancy, are useful for detecting single core errors or sense amplifier failures, or any channel failure that results in a single bit being in error. This concept extends to the detection of a fault within an arithmetic unit. For example, if an error is additively incorrect by a power of two, a residue code is useful, e.g., where the redundant digits represent the residue of the word modulo 3.

This approach also generalizes to the case where a fault produces an error in a single b-bit byte of memory or a channel, or in a b-bit byte of an arithmetic processor. In memory, b bits of redundancy suffice to detect errors in a single byte for words of arbitrary length. In arithmetic, similar redundancy is required (see section 5.1). The arithmetic codes may also be used in memory. The byte error situation

is a natural consequence of a byte-sliced memory or arithmetic unit, wherein each byte is realized as a single LSI chip.

Unfortunately, codes for detecting errors in a single bit or byte are not effectively extendable to arbitrary logic. (An exception is a function realized exclusively with linear logic.) One can, in principle, design a logic unit such that at every interface the vector of signals is an error-detecting code word in the absence of a fault, or not a code word in the presence of a fault. Given n independently realized output functions, the simplest way to do this is to provide a circuit with an output which is functionally the modulo two sum (parity check) of the n outputs (Lofgren 58). Thus faults producing an odd number of errors are detectable. However, this simple approach is not practical for the following reasons.

* For most practical functions, the semi-empirical results of Pierce (65) indicate that the cost of realizing the redundant function output may approach the combined cost of realizing the functions themselves. Thus, on a component count basis, this approach may be as bad as duplication.

* For nonindependent realizations of the n outputs, a single gate fault is likely to corrupt more than one output (e.g., an even number of outputs), especially in an LSI environment. Recent work by Ko (73) suggests possible circuit augmentations that ensure the corruption of only an odd number of outputs. This work indicates that some functions exist for which the total component count is less than that of duplication, but these functions tend to be exceptional. Besides, the relative saving seems to be insignificant in practice.

* Favorable results seem to rely upon the model of a single faulty gate with just one stuck-at-fault, which is, of course, not a reasonable assumption for MSI/LSI circuits.

* Finally, this approach is intended for multiple-output functions with the same set of inputs. For a single output, it reduces to duplication.

In general the nonapplicability of coding techniques for logic tends to reinforce the early pessimistic results of Elias (58) derived for a serial logic unit, showing the necessity of duplication for error detection in a single AND gate. The only exceptions to the apparent 50% redundancy are specialized functions. For example, a single gate in error in a tree-realized memory decoder results in either the selection of no word at all, or the selection of multiple words. If it is assumed that the accessed words are ANDed together (or ORed together) in corresponding bit positions, then a comparatively economical code can be used for error detection. Each n-bit word is encoded so that half of the bits positions contain a "1" and half contain a "0", e.g., the "n/2 out of n" codes. The redundancy is quite low (e.g., about 10% for 32-bit words, less for longer words). The encoding and decoding cost is small relative to the total memory cost, although it is higher than that for single error correction (Anderson and Metze 73). This code can detect arbitrarily many multiple errors if they are all of the same type (e.g., either all 0 to 1, or all 1 to 0).

One other coding scheme has been suggested for possible use in the PRIME system (Borgerson A2), to detect address decoder failures or memory bit-line failures. If a single error occurs in the address decoder, then a word will be accessed whose address is additively incorrect by some power of two. By noting the similarity with the effect of arithmetic errors, it is clear that this type of address decoder fault can be handled by appending to each memory word the modulo 3 residue of the address. Thus, this scheme detects any single error in memory or in the address decoder.

3.1.1.2. DUPLICATION

The essence of duplication is simple and straightforward. Results are independently computed twice, and the results compared. If the results are binary-valued, a disagreement indicates that one of the computations is in error. (If the results are multiple-valued, both may be in error.) The identification of the erroneous computation is deferred to

a more elaborate diagnosis. (A coding purist would contend that duplication really involves a trivial error detecting code. However, since there are interesting engineering details concerning the application of duplication at various system levels, it is worthwhile to discuss this apart from any coding theory implications.)

Although duplication is in principle applicable to any subsystem, it has primary application where less costly techniques are inadequate. That is, duplication is used for error detection where better techniques do not work. Generally, duplication may be used in conjunction with arbitrary logic in processors, I/O control units, special control circuits, and some memory functions. There is clearly little need to use duplication in conjunction with storage in main memory, except possibly in certain critical applications.

Duplication may be employed in SPACE (using two identical units) or in TIME. In TIME DUPLICATION, only one unit is used to perform the same computation twice (but perhaps internally reconfigured or shifted) before the computation is accepted as error free. Time duplication is less credible in that it depends on the equipment being exercised in different modes in order that the two computations do not agree because of identical or compensating errors. Variations and combinations of space and time duplication are also known. For example, two supposedly complementary versions of a result may be generated. For processors with an iterative structure, output data may be computed twice, but with permuted assignment over the identical modules.

The most obvious and common practice of duplication is to make a comparison on every machine cycle. On the other hand, if the comparison can be deferred, there may be an advantage to performing it in software. However, a software implementation requires the careful isolation of uncompared results to prevent error propagation. A software implementation also requires separate working memories for the pair of processors to hold intermediate (i.e., uncompared) results. Most fault tolerant systems employ hardware duplication to avoid the error propagation problem, but it is our opinion that careful attention to

recovery issues can lead to a feasible software implementation, at a substantial saving in hardware cost.

Another important engineering detail is the LEVEL OF PARTITIONING. The issue here is the identification of the interfaces at which comparisons are to be made. For example, the comparison interfaces can feasibly be at the system level (e.g., comparing the results of subroutines or procedure calls on exit), at the processor level (e.g., comparing two processors nominally executing identical instructions), and at the subprocessor level (e.g., comparing the outputs of byte slices of an arithmetic unit).

In most fault-tolerant systems, the error detection interfaces define the units to be removed in combatting faults. That is, if the arithmetic unit is a replaceable unit, then there usually exists some mechanism for detecting possible errors in the signals emerging from that unit. In any event, systems proposed for high-reliability, long-life applications typically employ a partitioning for error detection at a low system level. On the other hand, for most applications of concern here, detection at the processor level or memory unit level probably suffices. The roles of various levels in a fault-tolerant system are discussed in more detail in Section 3.2.

3.1.1.3. DEFERRED DETECTION

It is often not essential to detect a fault or an error as soon as it occurs. If the detection of a fault or error can be DEFERRED, it is possible to reduce the redundancy requirement for detection. Deferred detection may be performed COMPLETELY (deterministically) subsequent to the occurrence of a fault, e.g., on exit from a computational block. It may also be performed PROBABILISTICALLY, if over some period of time, there is a probability p that the error is detectable (e.g., in terms of a syndrome or other discrepancy). Three application areas of deferred detection are relevant.

NON UNIFORM DETECTION. In many fault tolerant systems, error detection facilities are applied uniformly to all processes. In many cases, errors can be allowed to occur without serious consequences, i.e., the errors are non-critical. We see a possibility for some economies in fault-tolerant equipment by applying error detection on the basis of criticality.

INCIDENTAL DETECTION. In some cases it is simply hoped that sooner or later (hopefully sooner) errors will be detectable without the use of much extra redundancy. This may be acceptable in low-cost units, or in cases in which the input state sequence is highly predictable.

UNFLEXED DETECTION. An output which changes only rarely from its nominal state needs special detection. A pertinent example here is a fault in the decoder for an error-detecting code that results in a constant "no-error" condition being emitted, even in the presence of errors. Similarly, certain system functions that are executed extremely rarely also require special detection. A latent fault in such a rarely used function could remain undetected and eventually result in a system failure. This problem is called the UNFLEXED-FUNCTION DETECTION problem. In general faults in such functions need not be detected as soon as they occur, e.g., because another hardware fault must occur before this function is required. Hence the detection of such faults can be deferred, i.e., carried out probabilistically.

An elegant theory has been developed to handle the third area (e.g., Carter et al. 72a, Anderson and Metze 72). For example, a conventional error-detection circuit might emit a "0" if there is no error, and a "1" if there is an error. Obviously, any fault that leads to a permanent emission of "0" will remain undetected. In order to alleviate this difficulty, two or more output lines are provided for the decoder. In the case of two output lines, a "0" might correspond to 00 or 11, and a "1" to 01 or 10. The decoder is designed such that when there is no error, the decoder on its two output lines emits 00 and 11 with equal probability. The decoder is designed such that any single stuck-at fault within it causes the output 01 or 10 for at least some code word

being presented at the input. This latter property has led this type of structure being called SELF-TESTING. The important positive conclusions from this work are the following.

* There exist fault-detection techniques that require less than duplication in implementation, although for incomplete detection. For the case of the decoder for a single-error correcting (Hamming) code, the redundancy is about 25% (Carter et al. 70a).

* There is an alternative to periodic diagnosis in detecting faults in unflexed circuits.

The negative conclusions are these:

* The redundancy requirements are low only for well-structured functions, e.g., decoders for error-correcting codes. For other unflexed circuits, duplication may be as good.

* The unflexed circuits represent a low proportion of total system cost. Thus, the incremental cost of using replication may be negligible.

* The fault model for the circuits is still concerned with single stuck-at faults. For more realistic faults, it is likely that duplication is close to optimal.

* Self-testing circuits appear to be a good solution for certain functions associated with the unflexed function problem. However, it is generally not clear that all such unflexed functions are attractive candidates for self-testing logic, when compared with periodic diagnosis and brute-force replication.

3.1.1.4 ERROR DETECTION VIA DIAGNOSIS

An approach to error detection that is potentially quite efficient involves periodic diagnosis of the fault-prone system blocks. A CHECKING SEQUENCE is imposed on the inputs of the blocks in question

such that if any fault is present an output value will eventually emerge that differs from the expected value. In order for the diagnosis approach to be effective, as compared with say duplication, the following features must be given consideration:

FAULT COVERAGE. Clearly the checking sequence must be capable of revealing an extremely large fraction of the likely fault patterns. Most research in fault diagnosis has been concerned with networks in which faults are manifested as a single gate being stuck-at-zero (SA0) or stuck-at-one (SA1). In an LSI implementation the single stuck-at assumption is not valid. An imperfection in an LSI chip tends to propagate outward from some source point. Thus it is likely that gates within a region will be suspect. It is likely that a checking sequence that handles all SA0 and SA1 faults will handle a large class of other fault patterns, although there is little formal work to substantiate this conjecture. With regard to non-formal work in this area computer manufacturers have developed checking sequences to help detect failures within their CPUs. Typically, these sequences are generated by ad hoc techniques and reveal only about 90 percent of the likely fault patterns. The conclusion here is that at present the fault coverage is not adequate for the error detection function. However, we feel that if the research effort is devoted to realistic fault models, and is coupled with simulation techniques this situation could be alleviated.

PERIODICITY OF CHECKING. The checking sequence must be applied often enough so that the probability of two faults occurring during the intervening period is low. Also since the faulty equipment might be unavailable during the inter diagnosing period this period must be shorter than the maximum tolerable unavailable time. For all but the critical real-time applications neither of these constraints is limiting. It is unlikely that a diagnosis of any system block needs to be carried out with a period shorter than 10-100 seconds.

DIAGNOSIS OVERHEAD. The important overhead measures of diagnosis are the amount of cpu effort devoted to diagnosis and the amount of high speed memory needed to store the checking sequences. Concerning the cpu

overhead the typical length of a checking sequence for arbitrary logic is one-tenth the number of gates. (This is our experience for the single stuck at model, but for more realistic models the length should not increase by more than a factor of two or three.) Thus a 10,000 gate processor can be checked with a sequence of length 1,000. Assuming 2 per test the total diagnosing time is 2 msec. The cpu overhead is thus negligible for an inter checking period of 10 seconds. For this inter checking period it is likely that the test itself can be stored on disk thus precluding the need for high speed storage.

ERROR CONFINEMENT DURING INTER CHECKING PERIOD. All computed results are suspect until the processor is diagnosed. Thus it is necessary to prevent possibly faulty results from propagating. In the PRIME system, which utilizes diagnosis as a primary error detection mechanism error propagation is not a problem because of inter processor isolation. In other systems the error confinement techniques of Sect. 3.1.2 must be considered.

FALLIBILITY OF DIAGNOSING SYSTEM. A paramount problem in diagnosis relates to the problem of faulty behavior in the system carrying out the diagnosis. In a system consisting of a single processor the best approach involves bootstrapping. Here a small system, assumed to be infallible carries out a diagnosis to verify the integrity of a larger system. This larger system then acts to produce a still larger verified system and so on. The initial small system can be made error detecting by duplication techniques. In a multiprocessor the commonly conceived approach is to have one processor diagnose another. If the diagnosing processor reports an error it is not decidable which processor is faulty. (If no error is reported it can be assumed that the diagnosed processor is operative, provided no more than one processor is assumed to be faulty.) If three or more processors are available, various strategies can be invoked to resolve the ambiguity.
(Preparata et al. 69) have presented one such strategy based upon a circular configuration of diagnosing processors.

In conclusion periodic diagnosis is potentially the most efficient

approach toward error detection. The only foreseeable limitation is the inapplicability to transient faults. For permanent faults additional work is needed to improve the fault coverage obtainable with checking sequences, particularly related to a fault model that is realistic in an LSI environment.

3.1.2. ERROR CORRECTION

The state of the art of coding for error correction and efficient (fast, cheap) decoding is well developed (e.g., Peterson and Weldon 72, Berlekamp 68). Error-correcting codes exist for use in memory and in arithmetic, for various types of errors. Such types include correction of single errors, independent multiple errors, and correlated errors (e.g., arbitrary errors within a byte, or confined to a burst of consecutive digits). Memory and arithmetic are covered in Sections 4.1 and 5.1, respectively. For error correction in processors and arbitrary logic, triplication and voting is the traditional technique. Many of the comments in Section 3.1.1 for error detection are also extendable to error correction.

3.1.3. RECONFIGURATION AND RECOVERY

Table 3.1 includes several items on reconfiguration and recovery which are fairly self-explanatory. As seen in Section 3.2, dynamically alterable strategies are needed, including instruction retry and recovery from a parity error in memory (depending on what word was in error, and what it was being used for). Reconfiguration of memory is discussed in detail in Section 4.2 and in Appendix 3. Recovery is discussed in Section 3.2.5.

3.2. STRUCTURED DESIGNS FOR FAULT TOLERANCE

Most computer system designs seem to evolve in an ad hoc fashion, reflecting both the structure of the organization(s) to which the designers belong (Conway's Law) and the lack of a holistic design view. Here we examine the role of structure in system design, and how it can facilitate the effective use of the above techniques for fault tolerance. (This section is inspired by Simon 62, Dijkstra 65, 68, 69, and Neumann 69, 72, 73. Also relevant is the work of Horning and Randell 73, and Parnas 72.) Well-conceived system structure can contribute significantly to the design, implementation, debugging, verification, testing, diagnosis, maintenance and operation of fault-tolerant systems. As employed here, such structure permits a wide range of techniques to be applied selectively and/or dynamically, when and where they are most effective in terms of cost and reliability. Low cost can be achieved by taking advantage of nonuniform constraints and various time-space tradeoffs. This is in contrast to many existing systems which employ (statically) primarily low-level techniques for fault tolerance. A well chosen system compartmentalization helps limit error propagation, improves autonomous maintenance, and enables the persistence of system security in spite of faults; it also facilitates long-term evolutionary growth of the system, responsive to new applications needs, new hardware, and new software.

Hierarchical aspects of such structure permit a hierarchical recovery strategy directly reflecting the structure of the design and the needs for recovery. Such a strategy can be relatively efficient, in that it can be dynamically tailored to the actual fault(s). Recovery varies widely in complexity, depending on the nature of the faulty behavior. It may be quite simple, as in the case of a detectable transient error in arithmetic (with buffered instruction retry) or in a memory with error-correcting coding, or it may be quite complicated, e.g., after a total collapse of the system. In general, the recovery strategy should assure recovery of the most critical parts of the system first. Structured recovery strategies are found to some extent in the Plessey System 250 (Williams A2) and in Multics (Saltzer A2).

The system design should integrate the needs for fault tolerance and for effective recovery with the other system needs of security, efficiency, capability, etc. (the PRINCIPLE OF GLOBAL DESIGN). Successful integration is greatly facilitated by a highly structured design that deals with architectural concepts irrespective of whether they are implemented in hardware, in microprogram, or in software, and which evolves in a roughly "top-down" or goal-driven fashion. Since software capability of one generation is frequently found in the hardware of the next generation, this view is highly appropriate.

3.2.1. STRUCTURAL LEVELS OF INVISIBILITY

The structure of a system can have considerable impact on the fault tolerance of the system, as well as on the system development as a whole. Although this subsection considers the role of such structure in general, it provides a basis for fault tolerance throughout this report. Of interest in this subsection are the interrelations that form the structure among the various system mechanisms. At the interface to each system mechanism, various implementation details may be hidden from the invocation of that interface. When an interface to a mechanism makes such implementation details invisible, that mechanism is said to be a virtual mechanism (see below). The interface provides a level of invisibility between its invocation and its implementation.

There are many different structural views of the mechanisms within a computing system, both system-oriented and user-oriented. The techniques for fault tolerance may be applied at various levels with respect to any of several such views. Consider first several system-oriented views. With respect to hardware dependence, levels of structure vary from components to subunits to functional units to a raw machine to a microprogrammed machine through various levels of software support to a network of systems. Corresponding levels of language capability (above the circuitry levels) range from microprogram instructions to machine language instructions to macro-assembly and compiler statements through various levels of block structure,

subroutine, and operating system calls, to system commands and network commands. A command itself may be substructured, with various levels of subrequests and requests within it. In units of time, levels of responsiveness range over a wide spectrum of response requirements, with different mechanisms requiring responses of picoseconds to nanoseconds to microseconds to milliseconds (e.g., for peripherals) to seconds (e.g., for human interaction), etc. Within a system, different sets of levels exist with respect to processors, memories, input-output, control, and intercommunication. In memories, for example, such levels range from storage for a bit of information to storage for (encoded) representations of words to blocks to memory modules to a hierarchy of diverse types of memories, e.g., managed (in hardware and software) as a single level of memory and organized into a directory structure (e.g., as directories of directories of files). In communication, levels range from intraprocessor communication to interprocessor, intersystem, and even internetwork communication. Other levels that are more or less orthogonal to the above levels are also distinguishable, e.g., the levels of reliability and protection discussed in Section 3.2.2.

A system in execution is controlled in hardware and in software by its OPERATING SYSTEM, and may be viewed overall as a collection of PROCESSES. Each process is a single locus of sequential control, relative to some address space. A process may invoke or create other processes, but in itself may not have multiple simultaneous "threads" of execution. Thus a process is the basic unit of asynchronous processing. Each process may be thought of as using a VIRTUAL PROCESSOR, i.e., a processor exclusive to that process. The address space of each process is its VIRTUAL MEMORY, with just that information (stored in a portion of actual memory) which is directly accessible to the process. The virtual memory provides a (simplified) interface to the real memory, and makes the management of actual memory largely invisible. The operating system may be thought of as multiplexing the various processes onto the actual system, and multiplexing the corresponding virtual memories onto the apparent single level of memory. At this level the mechanisms of MULTIPROGRAMMING (i.e., the concurrent use of main memory by several processes) are invisible. The operating system may itself be executing

in a MULTIPROCESSING mode, i.e., if it is able to run on multiple processors simultaneously. Some of the operating system processes may be allocated dynamically to special-purpose processors, while others may be permanently dedicated to specific hardware.

Each process also has facilities for input-output. Here levels range from data representations on devices and on media to data structures (e.g., bytes, characters, records, files) to various forms of VIRTUAL INPUT-OUTPUT (with invisibility of many details of device dependence, of multiprocessing and of multiprogramming, e.g., via virtual devices with invisible formatting and symbolic device attachments). The system is responsible for multiplexing the actual input-output devices and media.

There are various levels of process structure, from protection domains within processes, to processes within a system, to intrasystem and intersystem process families. From the view of a single "user" (whether he is a casual turn-key user, a systems program developer, or an environment being controlled by or controlling the computer system), he may see a single process. He may also wish to distribute a job among several asynchronous processes within a FAMILY OF PROCESSES. In the presence of multiple processors, this leads to multiprocessing at the user level. His process family makes many process mechanisms invisible. Each user has his own view of the actual system, which may be thought of as his VIRTUAL SYSTEM. (In some systems the process family view and the virtual system view may be identical.) Apart from inter-user communication and file sharing, a virtual system appears to each user as his own private system, and may be different (in part) from the virtual system of other users. A user may wish to invoke several virtual systems, either on one actual system or on several systems in a network. The simultaneous use by one user of different systems within a network leads to the concept of a VIRTUAL NETWORK, in which many details of system multiplexing are invisible.

Another user view arises with binding. BINDING refers to the act of reducing the indefiniteness of an incompletely specified entity (e.g, by assigning it a resource). Levels of binding specificity typically range

from program specification to program generation to compilation to object code generation to linking, loading, and execution. Linking and loading may each be partially static (in advance of execution) and partially dynamic (being invoked during execution with respect to other executing programs). At each successive (lower) level of binding, more machine-dependent detail is added to a program or collection of programs. This detail is normally not visible to the higher levels of binding.

A conceptually simple but highly powerful linear structuring of system levels is discussed by Dijkstra (68,69). Internal details of implementation at a given level are normally made invisible to higher levels. Functional capability at that level is dependent on the capability of the next lower level, and is precisely that provided by the lower-level interface languages. (That functional capability may in fact represent a loss of power compared with the next lower level.) The levels are referred to as LEVELS OF INVISIBILITY. Successively higher levels correspond to larger units of time. (In the sense that an interface creates a higher-level concept, it provides a LEVEL OF ABSTRACTION.)

More generally, a VIRTUAL mechanism is one that provides a layer of invisibility between the interface to that mechanism and the details internal to the implementation of the mechanism, independent of the structure among the various mechanisms. It may in some cases also reduce the power of that mechanism available at the given interface, but can in no way increase it. (Note that even a gate appears as a virtual mechanism to a logic circuit using it.) This does not mean that all details of the use of such a mechanism are invisible. In fact, efficiency considerations may dictate that some controls on the use of the mechanism must be accessible at the virtual interface (although not normally required). Similarly, it may sometimes be desirable (e.g., for efficiency) to use directly a mechanism at a more detailed level, rather than passing through many levels of interfaces. In some sense, most mechanisms can be viewed as virtual mechanisms. However, the PRINCIPLE OF LEAST VISIBILITY dictates that implementation detail should be

visible only where necessary. It is desirable that this principle have a strong influence on the structure of the system.

For any given set of virtual mechanisms, there is an interconnection structure among them by virtue of the use of their interfaces. Dijkstra's linear structuring of levels is not always realistic. In a complex system, the partial ordering among virtual mechanisms may be an arbitary directed graph, rather than a linear ordering. Nevertheless, there may be local regions in which it is linear or tree structured. In general, it is highly desirable to have a tree structure if not a linear structure. In some cases it may also be desirable to lump a collection of mechanisms into linear levels (e.g., for descriptive purposes or for implementation simplicity), even though these mechanisms are not properly linear. However, the extremes of excessively simple structure and excessively compartmentalized structure should both be avoided. It is extremely helpful to keep these types of levels conceptually distinct while designing a system, even if they are blurred in the resulting implementation, e.g., to achieve adequate performance.

3.2.2. LEVELS OF CRITICALITY

Given a structure among mechanisms dictated by the principle of least visibility, additional constraints arise in terms of implicit or explicit levels of criticality, e.g., sensitivity to fault-induced errors. The lowest (or innermost) levels (of highest sensitivity) are often referred to as the "hard-core" or the "kernel" of a system. It is worth noting, however, that usage and definitions of such terms are far from standard. Refer, for example, to Appendix 2. The term "hard-core" is used in at least three nonequivalent but overlapping fault-tolerance senses, (a) survival, (b) coverage, and (c) exposure. Consider respective illustrations of these three senses: (a) "that which must survive" (Wensley A2), or "that whose malfunction could crash the system" (Ulrich A2, and implicitly Saltzer A2); (b), "that which is covered by redundancy" (Avizienis A2); and (c), "that hardware which is irredundant" (Hopkins A2), or "that hardware (redundant or not) whose failure will produce undetected errors" (Carter A2). Note that (b) and

(c) are roughly complementary views. Also note the view in PRIME (Borgerson A2) that there is NO hard-core (undefined), because the supervisor can float from one processor to another. An earlier usage is "that which must function correctly"(Forbes et al. 65).

A functional sense of criticality is also found. For example, the "hard-core" paging software in a paged environment usually contains some programs (e.g., certain buffers and programs supporting paging itself) which themselves cannot be paged out. There is also software whose frequency of use dictates that it should remain in main memory for efficiency reasons.

In addition many levels of criticality with respect to system security are relevant here, including the integrity of the system itself and of resident files. The kernel for security may be thought of as that part of the system whose correct functioning is most critical to the uncompromised security of the system. A related concept is that of a SECURITY PERIMETER, i.e., a set of functions (programs, processes, etc.) within which system security may in some way be compromised, either by misuse or by malfunction. The security perimeter in the absence of faults seems to be significantly larger than is generally recognized. In the presence of faults, it may be very large unless the system is carefully partitioned. Guarantees of system security are desirable, at least in a probabilistic sense, both in the absence of faults (but in the presence of possible misuse) and in the presence of faults. Unfortunately, the kernels for reliability, for availability, and for security are not conceptually identical, even though most systems tend to lump them together.

3.2.3. SYSTEM STRUCTURE FOR FAULT TOLERANCE

Structured system design and structured implementation are developing arts that have immediate use in the design and implementation of systems with economical fault tolerance. Although further work is needed to make such structure an integral part of the design, rather than just good practice, the benefits are already considerable. Recent efforts in

this direction are found in structuring software implementations, e.g., structured programming (e.g., Dijkstra 69).

The notions of invisibility and criticality impose various constraints on the structure among system mechanisms. For the purpose of designing and implementing fault-tolerant systems, the most critical mechanisms should be carefully identified, and separated from similar but less critical mechanisms. However, the various views of "criticality" should be integrated. As noted above, the interactions among correctness, availability, and security are particularly strong. There are also strong interactions with critical mechanisms for reconfiguration, recovery, and restart following detected faults (or detected security violations), for interrupt handling and abnormal condition handling, and for on-line interactive maintenance. These critical mechanisms also cut across hardware-software boundaries. Table 3.2 provides an illustration of such critical elements affecting fault tolerance. The multiprocessor architecture of Section 3.3.3 illustrates an economical system using selective and dynamic redundancy for these elements.

As a general rule, the mechanisms of greatest criticality themselves should be well structured and small enough to verify and control. This enhances selective and dynamic usage of various fault-tolerance techniques, when and where they are most effective, whether implemented in hardware or in software. It also facilitates controlling system operation and recovery, and can further enhance the verification of correctness of the system design and its implementation, especially with respect to fault tolerance and security. In this way it is also possible to anticipate the effect of faults on system behavior (including security) and to tailor the design and the recovery strategy to the possible faults, their likelihoods, and their possible effects. Such design is particularly important if security is to be maintained despite faults.

Structure among virtual interfaces enters naturally into system design as follows, as a result of the above considerations. It is desirable that this design be driven more or less from the top down, although it is usually necessary to iterate up and down in order to assure that the

Table 3.2
CRITICAL ELEMENTS FOR FAULT TOLERANCE

MEMORY MANAGEMENT

Memory maps, e.g., page tables, device maps, associative memory maps
Memory contents, including critical data, contents of some registers, input-output buffers, channel control words and interrupt cells
Memory allocation mechanisms
Memory bootstrap recovery and reconfiguration

PROCESSOR MANAGEMENT

Memory fetches and address formation, including page relocation, and generation and validation of protection information
Receipt and interpretation of interrupts
Critical microcode, including interpretation and protection
Process creation, dispatching, and deletion
Interprocessor communication
Some exception handling
Primitive reconfiguration control, configuration sensing and setting
Primitive accounting and measurement facilities

INPUT-OUTPUT MANAGEMENT

Channel control, especially of shared channels
Certain media contents
Some exception handling

design converges to a suitable system.

(a) Various system partitions are established explicitly in the design, identifying virtual mechanisms and levels of criticality, responsive to the various overall system goals of correctness, availability, security, functional capability, capacity, performance, efficiency, etc. Effective modularization involves careful control of communication among mechanisms to help limit error propagation. Useful mechanisms are known for this purpose, both to avoid conflicts and to permit sharing of programs, data and control (e.g., Dijkstra 68, Spier and Organick 69, Holt 72, Baer 73). Except for deadlock avoidance, such mechanisms are conceptually clear-cut.

(b) Associated with these partitions are subpartitions for selective use of the techniques of fault tolerance, as well as possible configurations of these techniques and possible modes of dynamic reconfiguration of these techniques within and among the partitions. Successive levels of binding noted above may be useful points at which to bind fault-tolerance techniques as well (dynamically or statically).

(c) Analysis, simulation, verification, and operating experience should be used to study the relative effectiveness of these techniques under varying demands and of reliable algorithms for deciding how and when to switch among configurations. The suitability of the choice of partitions should also be evaluated. The exact boundaries among hardware, microprogram, and software should be established as late in the design process as possible. Mechanisms with high duty cycles should very likely appear in hardware or microprogram.

The applicability of relevant techniques for fault tolerance to various virtual mechanisms is illustrated by Table 3.3. The first column of the table identifies some illustrative interfaces. (Higher and less machine-dependent levels are toward the bottom of the table.) The second column gives examples of concepts invisible at (i.e., outside of) each interface. The third column gives examples of the techniques of Section 3.1. These techniques can enhance fault tolerance within each

Table 3.3
EXAMPLES OF TECHNIQUES FOR FAULT TOLERANCE
APPLICABLE TO A HIERARCHY OF INTERFACES

INTERFACES (examples)	INVISIBLE CONCEPTS (examples)	APPLICABLE FAULT-TOLERANCE TECHNIQUES (examples)
PHYSICAL VIEW (HARDWARE):		
Components, chips	Technology details, fabrication methods	intrinsicelly reliable technologies, good engineering, quality control, coding and fault-masking, replication.
Subunits	Board layouts, pin connections, timing	Conservative design, reliable connectors, environmental control; *Diagnosis, component replication, some coding, double-rail logic, replacement.
Units:		
Processors	Processor algorithms Degraded modes Address calculation, absolute addresses, associative mechanisms Bus control Interrupts Microcode	*Automatic instruction retry; *Replication, coding. *Logic via arithmetic, double-precision half-units. Bounds checking, descriptor validation, memory protection, coding and replication in address generation, coding and cross-checking in associative memory. *Alternate routes, coding, degradable priority mechanisms. *Race-free fail-operational interrupt design. *Microdiagnostics, validation of microcode; coding.
Memory units	Cache mechanisms Internal representation Internal configurations Device characteristics	*Automatic reloading. *Coding on memory contents; *Read after write at certain levels, hardware-checked descriptors and type information. *Reconfiguration around bad memory (via paging, de-interlace). Use of read-only memories to avoid overwrite and aid recovery.
Input-output devices	Media properties, device dependence	*Coding on contents of media and transmission. *Verification, checking, read and compare after write.
System	Configurations	*Configuration sensing and self-reconfiguration, powering on-off incl. spares), distributing and replacing power supplies.
PROCESS VIEW (HARDWARE AND SOFTWARE):		
Virtual processor	Multiprocessing by system-binding of processes to processors: processor dispatching Multiprogramming--multiplexing of processes onto the system: process scheduling, process isolation Array computing Multiplexing of microcode	Coding, handshaking on interprocessor communication, avoidance of interprocessor interference; *Replication of physical processors as a single virtual processor, voting as needed. *Configuration insensitivity via checked table-driving. *Explicit measures of permitted degradation per process. Safeguards on interprocess communication (vs. lost interrupts, blocked polling), avoidance of interprocess interference, intraprocess protection (rings, domains, master modes). *Reconfiguration and replacement within the array. Isolation of system microcode from user-alterable microcode.
Virtual memory	Multiplexing of virtual memories onto real memory, backup and retrieval, directories, device maps protection mechanisms	*Replication of critical data in various places in hierarchy, including reliable cheap backup store; *Automatic rollback. Redundant pointers in directory structure and file maps to permit fast recovery; Access control on files (e.g., write protection); The use of pure procedure to inhibit loss of critical data or programs and to aid in automatic rollback. Redundancy in interprocess and file protection mechanisms.
Virtual input-output	Multiplexing of I-O, virtual devices Exception handling Asynchrony, buffering, channel management	Handshaking to avoid loss of information; *Status information. *Device switchability, media replication. *Coding (e.g., redundant headers); *Flexible error handling. Race-condition and deadlock avoidance. I-O device, media, and data protection mechanisms.
USER VIEW (SOFTWARE):		
Process family (job)	Algorithmic parallelism Allocation of processes Multiprocessing by user	*Replication of virtual processors for a single process. *Independent computational checks (via possibly distinct processes) within a process family; *Automatic rollback.
Virtual system	Multiplexing of virtual systems, sharing of data System correctness Command interpretation	Inter-user protection (from the system and each other). *Controlled sharing (if any); Self-identifying descriptors. *Validation, evaluation of effectiveness and correctness. *On-line maintenance; Good compilers, diagnostics, debuggers.
Virtual network	Multiplexing of computer systems and their inter-communication	*Coding on intersystem communication, alternate paths. Intersystem protection mechanisms. *Detailed status of network control and network requests. *Human intervention (as a last resort) with good judgment.

Asterisks denote techniques particularly amenable to dynamic use. Almost all techniques are suitable for selective use.

interface, although the details of their implementation should be largely invisible at the interface. Almost every technique is suitable for selective use. Those techniques which also lend themselves to dynamic reconfigurability are indicated by an asterisk in the table. The dynamic control over reconfiguration of such techniques may be done internally at each level, as well as (under controlled circumstances) via the appropriate interface language. Reconfigurations within one level are often independent of those at other levels. Techniques for reconfiguration and recovery from faults are found within most partitions.

The choice of structure among and within virtual mechanisms may depend on the particular system specification. For example, simplifying assumptions (e.g., no multiprogramming or no multiprocessing) often permit simplified structure. Further, each mechanism of Table 3.3 may be scattered among hardware and software. Contemporary hardware typically exhibits a superficial modularity at the functional unit level, although usually not internally to the extent desired here. Multics (Saltzer A2), Project SUE at Toronto (Sevcik et al. 72), and Hydra for the C.mmp at Carnegie-Mellon (Siewiorek A2) are systems that exhibit good structure in their operating systems.

3.2.3.1. EXAMPLE OF A STRUCTURED FAULT-TOLERANT COMPUTER SYSTEM

As a simple example, consider a multiprocessing system of five processors, each normally allocated at any moment to a distinct process. At the VIRTUAL SYSTEM interface, each user (or application environment) deals with a command language interface to the system running under a process or process family. Each virtual system may in turn employ one or more (real) processes, either invisibly on behalf of the operating system or visibly on behalf of the user to exploit intrinsic parallelism, or to provide redundant (but possibly algorithmically distinct) computations. At the PROCESS interface, each virtual processor, virtual memory, and virtual input-output capability may involve fault tolerance techniques.

Table 3.4
EXAMPLES OF VARIOUS MODES OF USAGE

When? \ Where?	Uniform (in space)	Selective (in space)
Static (in time)	VP Triplicated, vote on each instruction ——— Distance d=3 in all Mp, SEC throughout	One VP triplicated, vote on each instruction Others simplex ——— d=3 for some of Mp, SEC; d=2 for the rest of Mp, SED
Dynamic (in time)	VP Triplicated, vote on request (e.g., on block exits) ——— d=3 in all Mp, SEC on request DED otherwise	VP Triplicated on demand, vote on request; Normally simplex ——— d=3 for some of Mp, d=2 for the rest; SEC or DED on request, SED otherwise

Note: VP=virtual processor (possibly replicated), Mp=primary memory, d=Hamming distance; S=single, D=double, EC=error correcting, ED=error detecting.

At the VIRTUAL PROCESSOR interface, a single process may be executed on just one processor or redundantly on different processors, with comparison or voting. However, there appears to be a single processor, exclusive to each process. The configuration might for varying periods of time include two of the five processors both executing replicas of the same process in a comparison mode, or three processors in a voting mode, or even in rare cases five in a voting mode. Internal details of such mechanisms should be mostly invisible to each process. These modes may vary selectively (e.g., only certain processors might be usable in a replicated mode) and may change dynamically (for example running simplex except when certain critical operating system functions are invoked). Examples at this interface are found in the upper half of each box in Table 3.4. Examples of systems possibly able to provide such flexibility include ARMMS (Martin A2), C.mmp (Siewiorek A2), and SIFT (Wensley A2).

At the VIRTUAL MEMORY interface, device addresses are invisible. There is often redundancy in the implementation of a virtual memory system, some of which is suitable for recovery and reliability. In systems in which memory files do not directly become a part of a user's virtual memory but rather copies are made into the virtual memory (as in 360/67 TSS), there is the redundancy of the duplicate. In systems in which files (e.g., segments in Multics) directly become a part of a user's virtual memory when being actively used, a virtual memory page may be found in various versions and in various modes of replication on various devices in the memory hierarchy. For example, in a paged environment, various instances of a given page may exist simultaneously in a cache-type memory, in primary memory and in secondary memory. If it is part of a procedure that is "pure" (unchanged by execution), then all instances are identical (barring errors); if it is data, the instances may differ if there is no write-through, or else may be identical. This natural temporary proliferation can be used constructively to provide checkpoints, thus greatly facilitating automatic rollback. It is especially useful with various instances of critical data. The redundancy may of course vary, depending on instantaneous needs. In any

event the recovery and rollback strategies must be carefully integrated with the memory hardware and software.

At the VIRTUAL INPUT-OUTPUT interface, many details of devices are invisible (e.g., formats, recovery strategies, and asynchrony). Fault tolerance can be increased by extensive use of table driving, providing possibilities for the use of coding in the tables, as well as isolating the handling of various devices.

At the PROCESSOR interface, the structure and the implementation of each processor are largely invisible. There may be several levels of invisibility inside this interface. As seen by an instruction, for example, automatic instruction retry and physical memory addresses are typically invisible. Selective replication and replacement are suitable for logic and arithmetic, with coding useful for arithmetic in some cases. Especially critical within this level is address generation, with respect to both security and reliability. Coding and replication are useful in assuring that addresses are correctly generated.

At the MEMORY interface, byte-slicing, coding and reconfiguration (discussed in detail in Chapter 4) are examples of fault tolerance techniques that are usually invisible to the effective address of an instruction. All three can benefit from selective usage. For example, different codes may be used in different portions or types of memory. It may even be desirable to have some memory (e.g., for use by critical operating system data) with greater redundancy. These techniques also may benefit from dynamic usage. One such approach to coding entails different uses of a particular encoding. For example, consider a code with Hamming (or arithmetic) distance 4 for single-error correction and double-error detection. When one error is known to be permanent, the code may actually be used to correct a second error (Siewiorek and Ingle 73). When the multiple error rate is high, the code may better be used for triple-error detection (accompanied by increasingly loud cries for help). Another such example is the use of a byte-error correcting code as a multiple-error detecting code when multiple-byte errors are suspect. Still another dynamic approach is the use of varying encodings

depending on usage, even possibly by changing the word length. For example, it might be advantageous to use different encodings for different types of information, e.g., for data and for instructions to be executed. This could be aided by a tagged architecture (e.g., Feustel 73, Miller A2). Note that there is dynamic (but hardware supported) duplication of data in memory in the Intermetrics Multiprocessor of Miller (A2). (Typical examples at this interface are found in the lower half of each box in Table 3.4.)

Similarly at the MODULE interface, multiple arithmetic or functional units tied to a control unit may be used in replication for fault tolerance, or in synchronism as in the ILLIAC IV for handling parallelism in computation, or independently. The first of these applications substantially increases reliability, while the others may substantially increase the computational throughput. Degraded but continued operation may be achieved with multiple or byte-sliced units, e.g., by invoking a multiprecision mode among reduced precision units.

Explicit structures of virtual mechanisms are now evident in a few recent computing systems, both in hardware and in software. For example, the Multics protection mechanism (Schroeder and Saltzer 72) provides successive linear levels of resilience to errors in hardware, software, and humans in its levels of protection. A spectrum of criticality exists with respect to faults. Only malfunctions (hard or soft) at the lowest software level affect the viability of the system. Others have diminishingly serious effects on the correctness of operation as the level increases, e.g., aborting one user's process, aborting one command, or aborting just one request within a command. As with hardware, software techniques for fault tolerance may also differ from level to level. An example is provided by the SIFT environment (Wensley A2), in which a wide range of real-time criticality is found among various tasks, and for which redundancy can be suitably configured to the task.

3.2.4. IMPLICATIONS OF STRUCTURED DESIGN

In this section we discuss the numerous benefits of the structured design approach. These include enhancements of reliability and computational capacity, and reduction of cost.

SELECTIVE AND DYNAMIC APPLICATION OF REDUNDANCY. A wide variety of techniques for fault tolerance can be applied, each where it is most effective and responsive to the needs for fault tolerance and computing capacity. Each configuration of fault-tolerance techniques can be dynamically altered, on the basis of the current usage of the system. (The reconfiguration may affect more than one level at once.) The net cost of system fault tolerance can therefore be reduced, especially if rarely used fault-tolerance techniques can be performed reliably in software. Considerable savings also result if occasional modest real-time delays are permitted (e.g., for diagnosis, recovery, and reconfiguration), further reducing the need for dedicated hardware. The typically nonuniform distribution of costs within a system also permits a reduction of the incremental cost of fault tolerance. Memory costs (including secondary storage) seem to dominate total hardware costs in a well balanced system, even in emerging technologies (see Chapter 6). Consequently, the relatively small cost of redundancy in memory (e.g., varying logarithmicly with word length for single-error or byte-error correction throughout memory) may dominate the incremental cost, even with replicated processors, but even more so with dynamic and selective replication. Dynamic and selective use of coding (e.g., Table 3.4) further reduces the cost of fault tolerance. A tagged architecture may be of significant help in this respect. Structured design also facilitates checkpoint mechanisms that permit varying degrees of rollback at different levels, as needed. On-site maintenance and diagnosis are also aided.

GRACEFUL DEGRADATION. In general, computing capacity not currently dedicated to fault tolerance is available for useful computing, assuming reasonable system balance. It is desirable to configure among pools of modules, functional units, processors, and systems. The multiplicity of

each pool should be large enough so that graceful degradation is possible (i.e., that the loss of any unit is not serious). This increases the overall system effectiveness, in terms of both computing capacity and fault tolerance.

SIMPLIFICATION OF THE DESIGN PROCESS. Well-chosen system structure can enhance each stage of system development (including designing, implementing, documenting, debugging, certifying, analyzing, maintaining, and modifying the system). At each such stage the notion of levels of invisibility permits issues of fault tolerance relevant to lower levels to be abstracted and analyzed, aiding in isolating any side-effects. Thus the structure serves as a useful model as well.

ADAPTABILITY TO ADVANCED TECHNOLOGY. Recent technological advances (e.g., LSI) significantly improve the cost-effectiveness of many of the techniques for fault tolerance. These advances should also stimulate new architectural directions, such as multiprocessors with considerable multiplicity, and distributed-logic and logic-in-memory designs. The latter case involves large arrays of small memory elements, each containing processing capability. These arrays could be organized into subarrays of subarrays, possibly with structures geometrically oriented toward the problem to be solved (cf. Kautz and Levitt 72).

APPLICABILITY TO FAULT-TOLERANT SYSTEMS. The structural approach seems particularly effective for large general-purpose systems. It also seems useful for many systems with some tight real-time constraints, for which selective redundancy can result in significant cost savings, compared to the uniform use of high-order replication.

Questions of overhead and reliability must be examined carefully. It appears that the overhead due to the use of structure can usually be kept small, except when fault-tolerance limits are approached. It is obviously desirable that the mechanisms for controlling reconfiguration must themselves be fault tolerant, thrash-resistant, secure, and reconfigurable. Interference problems and intercommunication must also be handled reliably.

In highly structured systems, there is a basic problem of [illegible] overhead of multilevel interpretation (i.e., of successive [illegible] many levels of language). This problem is alleviated by [illegible] certain low-level language constructs to be directly (and [illegible]) available from outer levels. Where explicit level crossing is necessary (as in protection mechanisms), the interlevel communication mechanism should be simple. Judicious use of hardware for such mechanisms is essential, as in the case of various associative shortcut mechanisms. In some cases it is also advantageous to reduce the number of conceptual levels in the implementation.

Various questions remain unanswered by this discussion. Can the tradeoffs among fault tolerance, computing capacity, cost, overhead, etc., be rigorously characterized? Under what circumstances is it desirable to reconfigure? What kind of limiting behavior occurs as computing capacity or fault-tolerance capacity is reached? What are the penalties associated with having too little or too much structure? What happens to the notion of the "weakest link", namely, those mechanisms to whose malfunction the system is most vulnerable? Can this notion be distributed among less weak links? How does it shift during reconfiguration?

Our assessment of the structured design approach is that it has the potential for providing highly flexible and economical fault tolerance without greatly compromising system cost, system performance, and system efficiency. Some qualities of structure are found in the current art, but full realization of this potential requires further development.

3.2.5. STRUCTURED RECOVERY STRATEGIES AND MASSIVE-TRANSIENT RECOVERY

One useful approach for effective recovery over wide ranges of faulty behavior follows the PRINCIPLE OF LEAST EFFORT (Zipf 49). It is desirable to structure the system so that subsequent to a fault, the availability of the most essential services can be restored as rapidly as possible, deferring (or overlapping with restored operating capacity)

that which need not be done immediately. In this way, it is possible for the system to recover by successive iteration, outward from the most critical mechanisms. (See Carter et al. 71a for a discussion of the recovery problem and the control of recovery. See also Williams A2, Saltzer A2, Stern 73, and Stern and Van Vleck 73.)

As an example of a specific problem that can be greatly simplified by the adoption of a hierarchical structure and hierarchical recovery strategies reflecting that structure, consider the "massive-transient" recovery problem:

> A correlated fault source (e.g., a power surge or a bolt of lightning) has left all units of the system suspect, perhaps introducing both transient and permanent faults. The problem is for the system to diagnose and configure itself back into a working configuration and to validate itself for correctness, all under its own control. Note that the software as well as the hardware must be considered suspect.

This problem is essentially a generalized fault-tolerance problem, where performance may cease temporarily during and just after the massive transient. It is also closely related to normal system initialization. Design structure and dynamic reconfigurability both aid greatly in solving this problem. One solution involves validating a correct configuration of hardware and bootstrapping upward from the lowest levels, until a satisfactory rudimentary system is obtained. This solution is aided by the use of a hard-wired non-volatile read-only memory which provides a basis of correct programs for recovery. Further help is offered if pure-procedure instructions in this memory can be executed directly, and if these programs operate only out of local memory at first. By working outward, valid portions of the system begin to emerge. Also useful for providing checkpoints may be cheap once-writable memories (possibly asynchronous to the main control). (Another approach is to try experiments on various configurations of the whole system.) Note that this problem may be intrinsically insoluble for a given system configuration. It may also be insoluble for the particular massive transient, e.g., because not enough operational

equipment remains to self-diagnose and configure a valid system, or even just to operate such a system. (Furthermore, more equipment may be required for diagnosis than for operation.)

3.3. ARCHITECTURES FOR FAULT TOLERANCE

This section describes some general architectural configurations for fault-tolerant computers. Our intention is to show how the variety of techniques outlined in Sections 3.1 and 3.2 may be applied to the design of complete economical systems with high availability and high degrees of correctness as desired. We do not delve deeply into the design of particular systems, but rather merely attempt to justify their fault-tolerance behavior. For each system architecture, we indicate an estimate of overall redundancy, reliability, and availability measures. We also give methods by which error detection and recovery can be achieved, and a general assessment of the system. Aplications for these architectures are considered in Chapter 6.

We examine various types of system architectures here. Section 3.3.1 considers simplex systems, that is, systems with a single instruction stream, but possibly with replicated processors. Section 3.3.2 considers multicomputers (including networks) and loosely-coupled multiprocessor systems. Section 3.3.3 considers strongly-coupled multiprocessors, e.g., with sharing of data in memory among processors. Most of the system types form the basis for systems surveyed in Appendices 1 and 2, although several types discussed here have not yet matured into prototype or even paper designs as yet.

Where fault tolerance is a design goal, it can easily be incorporated into the design. In general, however, it cannot be retrofitted effectively into an existing implementation. As indicated in Chapter 6, suitable architectures for fault tolerance exist for all common computational applications. For these applications, fault tolerance can be achieved by the exclusive use of hardware techniques, requiring little modification to the operating system. However, if the degree of fault tolerance is to be matched to the application needs, and is to

require less redundancy than that associated with replication, then much more reliance on software is needed. In particular, the operating system becomes significantly more complex, and perhaps represents a more likely source of errors than faulty hardware.

The credibility of a particular fault-tolerance concept is of great concern. In the aerospace environment, the general practice has been to design extremely simple and crudely replicated systems. This simplicity is a consequence of the demand for systems that are obviously reliable, and perhaps amenable to human error detection and reconfiguration. This demand has precluded the use of the less redundant (although more complex) fault-tolerance techniques described in this report. We feel that these better techniques will become more acceptable as the new technologies emerge, and as operating systems become more reliable. Advanced fault-tolerant systems, i.e., those with high availability, fast recovery, and low cost, will place high demands on the operating system.

3.3.1. SIMPLEX SYSTEMS

In this subsection we view a simplex processor system as one in which only a single central processor is present, or in which all central processors are intended to operate with identical instruction streams and data. The earliest conceptions of fault-tolerant systems were simplex systems, employing low-level redundancy techniques (e.g., in gates or registers).

3.3.1.1. REDUNDANCY ONLY IN MAIN MEMORY

The cost and unreliability of most contemporary systems are largely dominated by the main memory. (We exclude peripherals from the immediate discussion, since their effects can be readily decoupled.) Typically, the main memory is 50% to 75% of the total digital circuitry in a medium to large system. The main memory can be made reliable by techniques embodying varying combinations of error detection, error correction, block replacement, and chip replacement. The use of these

techniques can provide fault-tolerance ranging from a minimum of error detection in memory to completely autonomous error confinement, reconfiguration, and recovery in response to memory failures. With the use of these techniques, the memory is from 5% to 35% redundant, depending on the word length, byte length, and the desired degree of fault tolerance. There are two possible deficiencies associated with memory coding and block replacement.

*The memory is prone to faults in external equipment, notably power supplies. This problem can be alleviated by providing a separate power supply for each block or for each byte slice of memory.

* If only the memory is protected by redundancy, the unreliability of the system is decreased only by a factor of about 3. Hence some form of processor fault tolerance is still needed.

Memory fault protection is rapidly becoming a common practice. Most machines have a parity check option on main memory, and some newer machines (e.g., IBM's System/370) incorporate error correction at the bit level or at the byte level. Most machines with relocation hardware are capable of reconfiguration around one or more faulty memory blocks. This is a primitive form of graceful degradation in that main memory functions are either lost or taken over by secondary or paging memories, with an accompanying reduction in performance. However, we know of no working machines that achieve reconfiguration autonomously, subsequent to a detected error. Such reconfiguration is not difficult to achieve, and can extend the up-time of a system enormously.

3.3.1.2. REDUNDANCY IN MAIN MEMORY WITH PROCESSOR REPLICATION

The simplest approach to tolerating faults in processors is to use replicated processors and provide some mechanism for resolving discrepancies among their outputs. In one mode the processors are duplicated and the two instruction streams are synchronously compared before being accepted as correct. Any discrepancy can trigger a single-instruction retry in the hope that the fault causing the

discrepancy is transient. If the retry fails the first time, a more complex recovery may be invoked. Finally, if all retry attempts fail because the fault is permanent, autonomous or human diagnosis can be undertaken to identify the faulty processor. The system is then reconfigured to use just the good processor. In any event the duplicated processor scheme can clearly prevent an incorrect result due to a single faulty processor. With the inclusion of some diagnostic procedures it can provide a system that remains available in the presence of one faulty processor. Such a system, including the cost of memory coding, may have from 33% to 45% redundancy, depending on the dominance of memory in the system. Besides its relatively high redundancy, this approach has two operational deficiencies.

* Inadequacies in the current diagnosis practice preclude the use of this approach in the most exacting fault-tolerance situations. That is, most diagnostic programs are successful in handling no more than 90% of the fault possibilities. Thus subsequent to a processor failure, the system may not be successfully reconfigured as much as 10% of the time.

* The comparison of processor outputs, if carried out in hardware, introduces a few extra gate delays. In high-speed applications it might be possible to pipeline this comparison with other operations at the expense of extra circuitry.

If a higher probability of successful autonomous response to an error is required, then a triplicated or higher-order replicated processor can be used. The processors can then be operated in a voting mode or a dynamic voting mode if there are more than three processors. The processor and memory are approximately in balance if the memory operates with single-error correction and single block replacement and if the processor is triplicated. In this case the probabilities that the memory or processor exhaust their respective resources are roughly identical. Recovery in the case of a triplicated processor should still include a single instruction retry, before trying to restart from an earlier state, or before discarding the disagreeing processor. The major drawback of grossly replicating the processor is cost.

Triplication of processors with error-correcting coding in memory can have redundancy as high as 60%.

3.3.1.3. TRIPLICATED SYSTEMS

We discuss above the deficiencies due to the use of different redundancy techniques for the processor and memory. Another deficiency for certain applications is the need to modify the basic computer design. One simple way of avoiding these difficulties is to operate multiple computers (including their memory) in a duplicated or triplicated mode. The results are compared whenever information leaves a computer, e.g., to a channel. The comparison in this case is done at such a low duty cycle that software voting may be feasible. When a disagreement is detected, the backtrack can be to the beginning of a computation or to the last channel invocation. In this case the need for saving register states in order to achieve single instruction retry is avoided. Of course, the main drawback of a uniformly triplicated system is its redundancy, which exceeds 67%.

This single replicated virtual processor concept was used in the Saturn V guidance computer. It is a possible mode of operation in a version of SIFT (Wensley A2) which is stripped down to exclude multiprocessing, and forms the basis for a flight-control computer under consideration by NASA-Langley.

3.3.1.4. REDUNDANCY APPLIED OVER PROCESSOR PARTITIONS

Among the major drawbacks of the triplicated processor scheme and the triplicated system scheme are the following.

* After a single processor failure, all spare processor resources are exhausted.

* The crude redundancy technique does not take advantage of the unique structure of particular processor sub-blocks. Thus the redundancy is higher than it needs to be.

Under certain circumstances, an attractive scheme is to decompose a processor into sub-blocks and to apply redundancy techniques appropriate to each sub-block. For example in the STAR computer (Self-Testing and Repair, Avizienis A2) the following sub-blocks are identified: arithmetic unit, logic unit, and control unit. For a more powerful computer than the STAR, one could also include stacks, expanded register sets, and scratch-pad memories, for example. The basic fault-tolerance method is to detect an error at an interface to one of these sub-blocks, and if necessary, to replace that sub-block with a spare. Residue codes are used for error detection in the arithmetic unit (and in the memory), a 2-out-of-4 code for instructions, and duplication elsewhere. In all approaches of this type, there is the need for some overall executive within the processor to act as the ultimate arbiter of all detected errors In the STAR, the TARP (Test and Repair Processor) serves this function, and is itself triplicated. Note that the TARP really serves as a "smart" bus with all inter-block transfers passing through it.

The system can be as low as 40% redundant with a spare for each sub-block. Moreover the up-time can be extended by a factor of 10, because of the partitioning of the processor. One of the most compelling advantages is that no radical change is needed in the functional partitioning of the system. The major deficiencies of this approach are the following.

* Major internal processor delays may be encountered due to the TARP, a situtaion that might be alleviated by pipelining its operation with other units.

* In an LSI implementation the partitioning might not be appropriate. This is particularly true if the computational requirements can be met by a one-or-two-chip computer. However, in much larger installations MSI is likely to be used in the near future. With an MSI implementation, a relatively fine partitioning is feasible.

An early version of the SERF computer of Raytheon (Stiffler 73) employs a partitioning similar to STAR. However, the arithmetic-logic unit is

decomposed into bytes while external switching can provide a routing around faulty bytes. A partitioning this fine is appropriate only where effectively zero maintenance is required for long periods of time. The MECRA computer (Delamare A2) uses a variety of coding techniques within the processor. In addition, graceful degradation is achieved by program modification, using microprograms that remain correct.

Our conclusion with regard to partitioned processors is that technology advances have precluded their applicability for their originally intended application, aerospace. However, they appear useful for large processor installations, provided the delay problems can be solved.

3.3.1.5. MICROPROGRAM-ORIENTED PROCESSORS

Many small to medium size processors achieve a rich instruction set by microprogramming. As the availability of high-speed memories increases, it is likely that microprogramming will appear in all but the super-fast computers. Microprogramming is used to realize many complex instructions that otherwise would require special hardware. Thus the instruction unit can be simplified and many special logic boxes (e.g., floating-point hardware, multipliers, interrupt handlers) can be eliminated. The net result is a total computer in which only about 10% of the hardware is not a memory function. Straightforward coding techniques can be used for error detection or correction. In addition, with writable control store, the microprograms can be paged and routed from a failed memory block to an operative one, or in an extreme case, to a memory block in slower memory. Crude redundancy techniques can be used for the non-memory hardware, at comparatively little incremental cost.

3.3.1.6. PROCESSORS WITH DEFERRED, PARTIAL, OR PROBABILISTIC DETECTION

Most of the architectures discussed above aim at correctness for all computations and availability in the presence of single faults. However, in many applications correctness is essential only for certain computations, such as those involving security and file protection

(including address generation). We describe here a system wherein single hardware faults cannot result in a security breach. It would be preferable that the machine halt rather than propagate an error critical to its security. To achieve this safeguard, certain equipment (notably base, mapping, and relocation registers) must be protected against faults. Error-detecting codes can help here. In addition, the functions that read or modify access rights must be checked. This requirement can be accomplished by consistency checks or duplication in space or time. A more reliable but less elegant solution is to provide a special replicated hardware unit within the processor that would execute primarily those functions within the security perimeter. If this unit can be designed so as to consume a small fraction of the computational resources in an integrated non-fault-tolerant implementation, then a replicated minicomputer within a large processor might suffice to achieve fault tolerance.

The detection of only those errors that are in some sense critical is a special case of deferred detection (Section 3.1.1.3). Short of coding or duplication, many features can be included in a processor to enhance detection. For example, the use of a tagged (or descriptor-based) architecture (e.g., Feustel 73, or the Burroughs B5500) can be used as a valuable tool for detecting hardware errors. Any error that leads to a type violation (e.g., execution of data, or adding a floating-point number to a Boolean value, or an attempt to manipulate a capability) could be detected. The central problem with this technique is to protect the hardware that manipulates the tags.

3.3.1.7. CONCLUDING REMARKS ON SIMPLEX PROCESSOR ARCHITECTURES

As noted in Chapter 6, simplex architectures are the most prevalent today. They will probably remain common in the future, at least in super-fast systems. Efficient methods exist for designing fault-tolerant simplex systems. For example, a system that is 40% to 50% redundant can be correct and available in the presence of single faults. In a multiprocessor organization, this redundancy can be reduced by a factor of at least 2 for applications in which most

computations need not be carried out reliably, provided certain critical computations are reliable. That is, the multiprocessor organization discussed below is better matched to applying redundancy nonuniformly in time.

3.3.2. LOOSELY-COUPLED MULTIPROCESSOR ORGANIZATIONS

In this section we describe several multiprocessor architectures that exhibit economical fault tolerance. We assume applications for which the various tasks run substantially independent of each other, in separate memory blocks. As noted below, the absence of sharing and of strong interprocessor communications greatly simplifies the design of such fault-tolerant multiprocessors. Multiprocessors with strong dependence among processors (e.g., with shared use of memory) are considered in Section 3.3.3.

It is clear that multiple processors are effective for fault tolerance, for at least the following reasons:

* Processors and memory blocks represent good replacement units.

* When all resources (processes, memories, etc.) are operative, they are all kept busy doing useful work. As resources fail, the operative ones take up the slack with a loss in performance. Thus, the long-standing goal of a gracefully degraded system is readily achieved with a multiprocessor, except for the detection and diagnosis problems.

* It is possible, in principle, to achieve redundancy that is variable in time and space. For certain critical computations, several processor-memory pairs can operate in a replicated mode. Moreover, this replication can be modified dynamically in time.

We divide multiprocessor organizations into three types: fixed multicomputer systems, configurable multi-computer systems, multiprocessors with common memory. (Note again that shared memory is

discussed in Section 3.3.3.) Included are systems in which there is an active processor and a monitor processor, and networks of systems. We discuss fault-tolerance techniques available for these organizations.

These designs have not generally been suitable for efficient operation of a large-scale general-purpose interactive computing (e.g., a computer utility). Most such designs have been suggested for an aerospace environment. The main reason for the unsuitability of these designs to such applications is that hardware is not present to support sharing or flexible communication between error-prone processes executing in different processors or memories. Several of the designs permit interprocess communication, provided the processors all operate in a replicated mode. In a trivial sense the system then is protected against security breaches caused by single faults, but we do not consider this to be a satisfactory solution for, say, a computer utility. A more desirable solution is outlined below, in which replication is avoided. The omission in this subsection of hardware to support reliably controlled sharing is intentional. If the mechanisms for sharing are nonexistent or severely restricted, then a process going awry because of a hardware fault cannot cause damage outside its restricted domain. A satisfactory solution to the sharing problem in the presence of hardware faults does not exist, but the architectural conifigurations discussed in Section 3.3.3 seem to be a step in the right direction.

3.3.2.1. FIXED MULTICOMPUTERS

The primitive element of a multicomputer is a processor/memory combination. In such an architecture, the system can be protected against a processor going awry by enforcing an intercomputer security discipline. Moreover, since the primitive element is essentially a self-contained computer system, there is limited need for communication among the computers -- except for the purpose of handling error conditions or message handling involving the executive. An example of a fixed multicomputer is the Pacific Coast Stock Exchange system COMEX (Wallace A2). Multicomputer configurations also include computer

networks (e.g., the ARPA Network) -- see Kuo and Abramson (73). The desired fault tolerance for networks is highly dependent on the component systems, on the interconnection network, and on the applications. Networks are discussed in Section 6.3.

With the help of a flexible switching network between I/O devices and the set of computers, jobs can be assigned to an available computer. There is no facility for one computer to write in the memory of another. For example, the protection against the erroneous overwriting of a disk file is enforced by permitting only an executive to modify the switching network. The executive is run independently in each of two computers so that its operations are checked. Errors are detected either by a disagreement among executive computers or by any self checks incorporated within the individual computers running application programs. Any of the self-checks discussed for a simplex processor system could suffice here. The executive operating in a checked mode could diagnose a suspected computer. Note that this executive is running at an extremely low duty cycle, performing only job scheduling and infrequent error control. Each computer will have a resident operating system to perform such operations as loading and subroutine linkage. The redundancy of this concept is quite low (not exceeding 10%) as measured by the amount of hardware and software devoted to fault tolerance.

A minor augmentation of the technique could provide for the checking of the application programs if desired by the user. In this case the application programs are run in two or more computers. The local executive resident in each computer (pertinent for this application program) periodically reads the results for this program computed by other computers. Any disagreements can be noted in the memories for future disposition by the system executive. Periodically, the processors read from specified locations in the executive computer's memory to determine if they should handle new jobs, become an executive computer, or possibly disconnect themselves. The error control protocol discussed above is a simplified description of the SIFT system (Wensley 72). The ARMMS system (Martin A2) is also a multicomputer concept, but

incorporates all executive functions within an especially smart interface unit.

The multicomputer approach is clean, and should find application in environments where the computer system is a relatively small portion of the total mission cost, and the application program demands are known to be near constant. However, for other applications the notable disadvantages of the scheme are:

* Because there is little intercommunication among processors, each disposable unit (processor and memory) must be fairly large in order to represent an independently viable computer. Thus, it represents a large unit to be removed upon failure. The configurable multi-computer discussed in Section 3.3.2.2 represents a finer and more realistic partitioning.

* Assuming that the individual computers are larger than mini-computers, then multiprogramming within a single computer is desirable if the computers are to achieve reasonable efficiency. However, there is a problem of maintaining isolation between the processes being multiprogrammed. In the presence of faults, such isolation can be achieved only by using the relatively expensive techniques of a replicated simplex system discussed in Section 3.3.1.

* The system is too inflexible for variable tasks. For example, there is no way to vary the high-speed memory allocated to a job.

3.3.2.2. CONFIGURABLE MULTI-COMPUTERS

We consider architectures in which a set of computers can be configured out of a collection of processors, memories, and (possibly) I/O controllers. The configuration is accomplished either manually by an operator or by an executive (in hardware and/or software). To accomplish such variable interconnections among system units, the system requires an interconnection network (e.g., a cross-bar or restricted cross-bar) between a set of memories and a set of processors, and

another such network between the I/O controllers and the memories. (Some switched communication links will also be required between the processor and I/O.) To inhibit deleterious error propagation from a failed processor, the interconnection network is changed only infrequently, e.g., when a new job is loaded in, or possibly only when a unit fails.

The fault tolerance procedures for a configurable multicomputer are almost equivalent to those of the architectures discussed in Section 3.3.2.1. For example, the CLC computer of Bell Laboratories (see Ridgway A2) uses a variety of consistency checks to detect errors. The PRIME system (Borgerson A2) relies on memory parity, periodic diagnosis, and user complaints to detect errors. There is no attempt to perform error correction on the above systems, so that the main forte of these systems is availability.

The redundancy is slightly higher than that for a fixed multi-computer architecture, mainly because of the need for extra hardware in the interconnection networks, and extra software to implement the more advanced reconfiguration possibilities. However, aside from the cost of spare units, the system should not be more than 15% redundant. The system is somewhat prone to faults in the switching network. Nevertheless, the effects of a single fault in the switching network can be made equivalent to a fault in a processor, memory, or I/O controller by distributing the switches among the units. If there is a need for certain computations to run concurrently in two or more computers, such an allocation can be effected by the executive. At the conclusion of the computation, the executive can gain access to the pertinent memory modules to compare the results.

The performance of a configurable multiprocessor is better than that of the fixed multicomputer in several respects:

* The configurable multicomputer offers longer life for a given number of spares, because of the finer partitioning. That is, when an error is discovered, a subsequent diagnosis can pinpoint the fault to a memory or

processor unit. In the fixed multicomputer, an entire processor/memory pair is discarded.

* The configurable multicomputer offers the possibility of adjusting the main memory requirements to the needs of a job.

* By virtue of the interconnection networks, there is the possibility for some interprocessor communications. However, the reliability needs dictate that this communication should be under the strict control of the executive.

Despite the above advantages, a configurable multiprocessor with the present state of the art does not meet the requirements of many computer utilities. This is true primarily because of the difficulties of achieving multiprogramming within each processor, and of achieving reliably controlled sharing of memory among processors.

3.3.2.3. LOOSELY-COUPLED MULTIPROCESSORS WITH COMMON MEMORY

For applications in which most of the computations must be fault tolerant, and in which there are real-time constraints on the computations, the several multicomputer architectures discussed above are grossly redundant. That is, the aforementioned multicomputers require that the computation be executed in two or more full computers. This fault-tolerance procedure does not take advantage of the low-cost coding techniques for memory.

Memory coding techniques can be used for both error detection and correction, as follows. The main memory is either a large block-oriented common memory or a multiport memory that can communicate with other system units by means of an interconnection network. Each processor unit is a pair of processors that will operate in a lock-step mode. Processor errors are detected by a disagreement between the processor outputs. To ensure that erroneous information does not emanate from the processor pairs, it is necessary to suspend the operation of the pairs when the error is detected. This can be achieved

by some reliable logic (usually triplication) at the interfaces to processor pairs and other system blocks. This approach has been taken in the Hopkins multiprocessor (Hopkins A2) and the Intermetrics multiprocessor (Miller A2). Another approach is to make the interconnection network powerful enough to isolate a processor pair in error. This approach is pursued in the BUCS system (Wensley et al. 73). In either case, a processor pair is discarded if the fault is permanent.

As mentioned above, system memory can take the form of a common memory or of a set of memory modules. In either case the memory information is protected with coding that provides at least single-byte error correction. When an error is detected in memory, the block or module in error is kept in service long enough to transfer its data to an operative section.

This concept is less costly than the multicomputer structures if correctness of results is important. The actual cost varies with the number of units needed to meet the computational needs, but typically the system will be about 50% redundant with one spare unit of each type. Moreover, these concepts can be extended to allow process sharing, since each processor's operation can be checked. However, this checking still requires duplication of all processors -- a cost that is not attractive for general use.

A common use of a multiprocessor configuration is where one processor is checking on the performance of another or doing background work, but is prepared to take over active performance. Such systems include No. 1 ESS (Ulrich A2) (with a monitor processor running diagnostics), and the New York Stock Exchange Market Data System MDS-2 (with two processors multiprocessing and a third acting as a monitor).

3.3.3. STRONGLY-COUPLED MULTIPROCESSORS

The multiprocessor systems discussed in 3.3.2 are primarily intended for the aerospace environment, or an environment in which processes can function independently. In the latter case, the multiprocessor

structure offers high availability. Sharing is possible only if all computations can be generated to operate error-free, which in turn generally requires costly replication for those multiprocessors. However, in a modern computer utility, controlled sharing is extremely desirable. Moreover, it should not be necessary to replicate entire processors in order to achieve reliably controlled sharing when the programs desiring sharing need not be error-free.

A useful example is provided by the Multics system. Among the important currently implemented features of Multics that bear on sharing and fault tolerance are the following:

* The ring structure (within a process) prevents a program (running in some ring) from disturbing a program that runs in some inner ring. In particular, an application program cannot crash the operating system.

* The operating system itself is layered with the security-dependent functions clustered in the innermost ring. At present that ring is too large for our purposes--an issue considered below.

* The file system is fairly immune to system crashes.

* Processors or memory modules can be added or deleted while the system is in operation.

Aside from the third item, these features are also included in the design for the SUE system (Sevcik et al., 72). Multics does little to support fault tolerance (e.g., there is at present no instantaneous attempt to recover from a parity error in memory), although there are substantial mechanisms for the integrity of resident storage. Under hardware faults, the only guarantee is that the system can eventually recover, with or without operator intervention.

The desired characteristics for a system embodying both sharing and advanced fault tolerance are the following:

* Sharing and protection are desirable, in the spirit of Multics.

* The protection mechanisms should not be violated under single fault occurrences.

* Processes should be able to execute on an unreplicated processor and still enable the protection mechanisms to be maintained.

* If correctness is needed for certain computations, then such computations should execute in a replicated mode.

* The individual processors should be multiprogrammable.

* Availability should be achievable by the inclusion of spare modules.

The Plessey 250 system (Williams A2) comes close to meeting the above characteristics. It is a multiprocessor structure with special hardware within a processor to support a capability-oriented protection scheme. Any process can invoke the operating system, so that the operating system as a part of any process on any processor. The detection of errors and the prevention of error propagation beyond a processor is achieved by combination of consistency checks and special self-tests within a processor. For example, a process accessing a segment for which the capability does not exist would cause an error indication. For the most part the Plessey 250 system operates in a benign environment, so that the capability checks are present mainly for error detection and confinement rather than for bootstrap recovery. A well-designed hierarchical recovery procedure is provided. The system is quite economical--less than 25% redundant and the error detection and recovery procedures have been evaluated by simulation. However, the system still relies primarily on ad hoc error detection procedures. If these design techniques are applied to a less predictable computing environment, there is no assurance that errors will be caught before they cause a crash or a security breach, nor is there any assurance that the recovery can be carried out autonomously.

It is possible to achieve all of the goals by performing certain operating system functions in a replicated mode. The Carnegie-Mellon C.mmp multiprocessor system (Siewiorek A2) offers some possibility in this direction. Briefly, the C.mmp is a multiprocessor in which a set of processors communicate with a set of memory modules via a crossbar type network. Certain interconnections can be inhibited by manual control of the network. Aside from this manual override the crossbar is settable by a block address generated by a processor. The contents of a set of mapping registers associated with each processor determine the capabilities of the process running in the processor in question. These registers can be set only by the operating system.

The most significant aspect of the operating system is its kernel, called Hydra (Siewiorek A2). Within its boundaries Hydra contains sufficient routines to enforce various protection and sharing disciplines among processes. Hydra also offers facilities for writing an extended operating system. Any process can invoke Hydra on its behalf. From a fault tolerance standpoint, all of the features presently in Hydra should be protected. That is, hardware faults should not induce any errors in the operation of the kernel. In addition to the current functions of Hydra, the reliability kernel should contain procedures for recovery, diagnosis, and configuration. Much of I/O does not belong in the reliability kernel except possibly a disc manager. It is intended that the reliability kernel be run in a checked mode. The most convenient way of achieving this checking is to run the reliability kernel, when it is called, simultaneously on two processors. If the memory modules incorporate their own fault tolerance (probably by means of error correcting codes), the two distinct memories are not generally required. However, since the temporary storage memory requirements of the kernel are small, each replicate of the kernel can run simultaneously in its own processor and memory. At a time when the kernel can return values, the process calling the kernel can read the results simultaneously from both memories, and can compare the results.

A minor hardware augmentation of C.mmp is required here. If this comparison and the resultant storage of the kernel results are to be

carried out reliably, then these operations themselves must be checked. One way of achieving this reliable abstraction of the kernel's computations is to expand the capability register set associated with each processor into a small duplicated microprocessor. The register set need not be duplicated, but can be protected by a simple parity code. This duplicated microprocessor (distributed among the processors) can be viewed as a distributed TARP or bus checker. It is also necessary to provide fault tolerance within the interconnection network, e.g., trivially by replicating the network, or better by distributing the network among the interconnected modules. In this latter approach feedback can be used to verify that control is established correctly.

In conclusion a multiprocessor structure like C.mmp or Plessey 250 can be extended so as to achieve all of the prescribed design goals at a comparatively low additional hardware cost. The addition of the duplicated microprocessor and the extra cost of fault tolerance in the interconnection network should be equivalent to about 20 percent of a processor. There is also the additional overhead of executing the reliability kernel in two processors--a cost that is presently unknown but should be low.

CHAPTER 4. MEMORY ORGANIZATION

This chapter describes the use of redundancy and reconfiguration in memory to increase system fault tolerance. Several of the better known schemes are treated, and a new approach is given that offers great improvements in availability for large memories.

In most of the systems of interest here, there is a diversity of memory types, from very fast small special-purpose memories (e.g., a cache, or associative memory for paging, or a microprogram control store) to fast main memories to various slower on-line memories (possibly block oriented) to normally off-line storage. A virtual memory mechanism is very helpful for the management of such a storage hierarchy, and can contribute to economical fault tolerance in several ways. First, by isolating real memory addresses from user programs, it contributes to security, especially if the address manipulations are done reliably. Second, it simplifies internal reconfiguration, replacement, and removal via page relocation, increasing operational continuity in the presence of faults. Third, it can provide a natural proliferation of different versions of data and procedures that can be very helpful in recovery.

4.1. ERROR DETECTION AND ERROR CORRECTION IN MEMORY

The coding art is well developed with respect to realistic codes and decoding procedures (e.g., Berlekamp 68, Peterson and Weldon 72). Thus this section presents various conclusions based on this art, as well as summarizing various aspects of byte coding for byte-organized memories.

As noted in Section 3.2, there is a wide range of criticality among various memory usages. Simple single-error detection or byte-error detection may suffice for much of memory. However, certain computations for which rollback is both difficult and undesirable may require error correction. Further, even where recovery is possible, some more reliable memory may be required. Fortunately, coding in memory is relatively cheap, even byte-error correction (see below), and especially

if used selectively.

Various special memories have special needs for coding techniques. For example, error correction may be neither desirable (because of decoding delay) nor necessary in an associative memory for which there is write-through or easy restoration of faulty words. However, capability for error detection may be very critical. For example, an error in the associative memory of a paged system can drastically affect both the system and its security. Burst coding (e.g., Elspas and Short 62, Elspas et al. 62, Berelekamp 68, Peterson and Weldon 68) may be effective in devices with serial transfer.

BYTE-ERROR CODING

Byte coding is highly appropriate for byte-per-chip memories, as in an LSI chip storing b bits from each of y words (e.g., b=4, y=1024). Here y n-bit memory words are stored in n/b chips. In some technologies it is possible for a fault to result in as many as b bits in a chip being in error, and thus byte detection or byte correction may be appropriate for the b-bit bytes.

Detection of a byte in error within a word with k=n-r information digits requires exactly r=b redundant bits, i.e., n=k+b, with b interlaced parity checks. Almost complete byte-error detection is achieved with the same redundancy using residue codes, which have the advantage that they are also useful for detecting errors in arithmetic (see Avizienis et al. 71, Parhami and Avizienis 73). Note that the same redundancy (and in fact the same code with interlaced parity checks) also provides BURST-ERROR DETECTION for burst errors, i.e., up to b errors confined to b consecutive positions (cyclic or otherwise) (e.g., see Peterson and Weldon 72). This is true even though b-bit byte errors are a subclass of b-bit burst errors.

Byte correction can always be achieved with generalized base B Hamming codes with $B = 2^b$. The redundancy (in bits) of these codes is

$$r = b \left\lceil \log_B \left\{ \frac{n}{b}(2^b - 1) + 1 \right\} \right\rceil , \tag{4.1}$$

as long as $r \geq 2b$; $\lceil x \rceil$ denotes the smallest integer containing x. Fewer redundant bits actually suffice in many cases, with a lower bor given by

$$r \geq \log_2 \left\{ \frac{n}{b}(2^b - 1) + 1 \right\} \text{ and } r \geq 2b . \tag{4.2}$$

This follows from the required number of distinct error patterns (each requiring a distinct SYNDROME, or check pattern) for each of the n/b radix B digits. The best codes known are those of Hong and Patel (72): if r is written as r=ib+c, with $0 \leq c < b$, and i an integer, then the value of k for a particular value of r is given by

$$k = b \frac{(2^r - 1) - 2^b (2^c - 1)}{2^b - 1} + c - r . \tag{4.3}$$

These codes are shown to be maximal when c is 0 or 1 (in the sense that no such code with greater k can exist for that r); Hong and Patel conjecture that this is true in general. The redundancy of these codes is often identical to the bound in (4.2). Since b=1 corresponds to the binary Hamming (single bit) error correcting codes, for which

$$r = \left\lceil \log_2 (n+1) \right\rceil ,$$

byte correction requires roughly $b - \log_2 b$ bits more than (single) bit correction. Note that b-bit (cyclic or non-cyclic) burst error correction requires

$$r \geq \log_2 (n\, 2^{b-1} + 1)$$

bits of redundancy, which is typically at least $\log_2 b - 1$ bits more than byte correction -- cf. (4.2).

Table 4.1 summarizes the redundancy of the Hong-Patel codes for typical values of k and small byte length b. Note that some byte-correcting codes with b=2 have the same redundancy as the Hamming codes for b=1, e.g., those with k from 28 to 36, for which r=6. The code with b=2 and k=36 is perfect, i.e., every non-zero syndrome corresponds to a distinct correctable byte error. (So are all of the Hong-Patel codes with c=0, corresponding to generalized Hamming codes.) Also noteworthy is the

perfect code for b=4, k=60, r=8. Finally, as a simple illustration of the gap between (4.2) and (4.3), consider the case of b=2 and k=15. Here r=5 satisfies (4.2), but r=6 is required for this case. The values of k shown are meant to be illustrative. If tag bits are included in memory words (e.g., Feustel 73) and encoded, the actual value of k (as opposed to its virtual value seen by data) may be quite unusual (e.g., 51 as in the B5500).

Table 4.1
SMALLEST POSSIBLE REDUNDANCY r FOR BYTE-ERROR CORRECTION IN MEMORY WITH VARIOUS BYTE SIZES b

Typical length k	Redundancy r for b= 1	2	3	4	5 .. 8
16	5	6	6	8	10 .. 16
24	5	6	7	8	10 .. 16
32	6	6	7	8	10 .. 16
48	6	7	8	8	10 .. 16
64	7	7	8	9	10 .. 16
128	8	8	9	9	10 .. 16

The cost of redundant storage for byte-error correction is thus seen to be relatively small for b=2 and 4 (even more so if used selectively). The cost in time delay can also be small. In fact, if automatic instruction retry is available, the cost in time can be effectively zero. This is possible for systematic codes (for which the information digits are directly available in a correct word--as in the case of Hamming codes), by overlapping the syndrome generation (i.e., error detection and implicit location) with the instruction execution up until (but not beyond) the point of no return for instruction retry. As long as syndrome generation completes before that point is reached, there is no delay at all due to decoding -- assuming no errors. (This requires pipelining the decoder in a pipelined machine.) If the word from memory contains an error resulting in a nonzero syndrome, the instruction

execution is interrupted, the word rewritten (correctly, after error correction) in memory, and the retry mechanism is triggered. Thus there is a delay only when an error needs to be corrected.

Various efforts are devoted to designing fast decoders (e.g., Bossen 70, Hong and Patel 72, Carter et al. 72b). Speed may also be enhanced by the use of read-only memories in decoding (e.g. Laws 72, Mitarai and McCluskey 72), both for the syndrome generation and for error correction, as well as by performing various manipulations on the parity check matrix.

The reliability of decoding for error correction may be enhanced by a technique of Kautz (62), in which redundant syndromes are calculated, providing a check on the syndrome generation itself. Such techniques are economical, especially since no redundancy is added to memory, and since the cost of the decoder(s) is small with respect to the cost of memory. Distributing decoders among memory controllers, or even memory modules, may have advantages of continuing availability of the system despite malfunction of one decoder. Such distribution also facilitates the selective use of coding, by permitting different encodings for different portions of memory. However it means that the busses are not checked. See also Carter et al. (70b) for self-checking decoders.

4.2. MEMORY RECONFIGURATION

In this subsection we consider schemes for reconfiguring a memory. The memory is assumed to be built from a number of units (for example, LSI chips) each having the same memory capacity. When a fault is detected, at least one unit is discarded and is either replaced by a similar number of spare units, or the system now has reduced memory capacity. We use the terms as defined in Section 4.1, with the following additions.

m = total number of memory units (e.g., LSI chips).

x = number of memory units in a block, i.e., the number discarded when a fault occurs.

y = number of words per block.

w = the total number of words in the memory

w' = the number of usable words required (<w).

We are concerned with two measures of reliability. By $P[\geq w':w]$, we mean the probability that given w words originally, there are at least w' words remaining at the time of consideration. The second measure used is $P[f]$, the probability that f faults can be tolerated. Schemes for memory reconfiguration are assessed by the above two factors, plus a measure of the cost of achieving the fault tolerance. We note further that the probability $P[f]$ of being able to tolerate f faults is irrelevant for some memory structures. Consider, for example, a block replacement scheme. All faults can be removed from the memory, although with a reduction of memory capacity. The single measure $P[\geq w':w]$ is therefore a sufficient measure of reliability for such a scheme. In some other schemes, to be described below, the switching capability is restricted and $P[f]$ becomes a meaningful measure of the ability of this switching network to remove faulty units.

Given x memory units in a block, the probability P_f of failure of a block is given by

$$P_f = 1-(1-p)^x \tag{4.4}$$

The number of blocks is u = m/x = w/y. The probability P_{fi} that i blocks contain faults is given by

$$P_{fi} = \binom{u}{i} P_f^i (1-P_f)^{(u-i)} \tag{4.5}$$

We use the notation $\lfloor a \rfloor$ to denote the largest integer contained in a. The probability that at least w' words remain, given w words originally is given by

$$P[\geq w' : w] = \sum_{i=0}^{\lfloor (w-w')/y \rfloor} P_{fi} \tag{4.6}$$

4.2.1. MEMORY RECONFIGURATION BY BLOCK REPLACEMENT

Consider a memory in which reconfiguration is carried by discarding the block in which a fault exists. We further assume that the switch network that carries out the reconfiguration can handle all fault patterns, ie the ability to reconfigure is not constrained by the switch but only by the availability of enough fault-free blocks. The reliability of such a scheme is represented by (4.6) above.

4.2.2. THE USE OF CODING WITH BLOCK REPLACEMENT

When coding is used for error detection and/or correction, as discussed in Section 4.1, it becomes natural to constrain the parameters y and b. The number of bits per chip yb is determined by the prevailing technology (values from 256 to 4096 are currently common). With too low a value of b, the number of data lines to the chip tends to make the number of words in a block large, causing discarding of an excessive number of words in the event of a fault. Too high a value of b has two bad effects. First, it increases the number of pins required for data on the chip. Second, if a code is used to detect and/or correct errors on a chip, then the coding complexity rises. We therefore have the possibility of tradeoff, which is analyzed in detail in Appendix 3.

Consider a memory constructed using LSI chips, in which coding is used to correct errors, and blocks of memory are replaced immediately after a fault occurs. The analysis in Appendix 3 assumes that a byte-error correcting code is used. The number r of redundant bits is related to k (the number of information bits), as discussed in Section 4.1.

Several detailed design topics are addressed in Appendix 3, particularly analyses of the optimum value of b, and the value of $P[w':w]$, given the probability p of chip failure.

The following conclusions are relevant here.

* Block replacement strategies for long-life use (i.e., p = .1) require

very high redundancy to achieve useful system success probability. Other fault-tolerance techniques should be used.

* For values of $p < 0.01$, the optimum value for b is 4 in almost all cases.

* For mission times of the order of a month or less (i.e., $p < .001$), very high values of $P[>w':w]$ can be achieved with less than 50% redundancy.

* One advantage of block replacement is that the memory chips do not need to contain special switching capabilities, as in some chip replacement schemes. Another advantage is the simplicity of the reconfiguration strategy. The disadvantage of block replacemeet is that it is very inefficient in its use of spares, in that nonfaulty chips are discarded because they are associated in the same block with a faulty chip. We must therefore consider schemes in which the unit of reconfiguration is smaller than the block.

4.2.3. RECONFIGURATION BY CHIP REPLACEMENT

A typical problem in a chip-replacement scheme is the cost of the switching network required to replace faulty chips with spares. The novel scheme presented by example below, and in general in Appendix 3, examines the possibility of economical switching for reconfiguration at the chip level. The primitive element in the memory is an LSI chip that realizes a section of memory b bits wide by y words long, together with an address decoder for the y words. The chips (including spares) are connected via a switching network so that the memory can be reconfigured effectively in the presence of chip failures. The main results relating to the switching network are as follows:

* The extra cost of the switching network and of the spare chips is low, compared with a nonredundant memory system.

* There are well-defined tradeoffs among the cost of the switching

network, the number t of chip failures be tolerated, the number s of spare chips, and the complexity of setting up the switching network.

* The switching networks can be embedded within the memory chips, so as to increase the reliability and increase uniformity.

A TWO DIMENSIONAL SCHEME

Consider first a non-reconfigurable LSI memory as shown in Figure 4.1. Each LSI chip contains d bits of b words and a decoder for the low order bits which are routed to all chips. The high order address bits are decoded to provide activation of one control line which selects the row of chips that contain the desired word. Data is routed to or from the chips via data lines shown vertically in Figure 4.1.

In the reconfigurable scheme to be described, the chips are augmented by the incorporation of two switches as shown in Figure 4.2. One switch enables the chip to be activated by one of three control lines from the decoder or to be made inoperative by setting the switch to the null position. Similarly the data switch can be set to be connected to either of two data lines or to a null position. As in the non-reconfigurable memory the chip contains a decoder for the low order address bits. The chips are assembled into a memory structure as illustrated in Figure 4.3 which shows that the control lines are connected to three rows of chips and each data line is connectable to two columns of chips. It is assumed that wrap-around occurs both vertically and horizontally, ie., the leftmost data line is also connected to the rightmost chips column, and similarly for the top and bottom control lines. An extra column of chips is provided that can be regarded as spares and which we will in this discussion regard as being the rightmost column. Each spare chip is controlled independently.

In discussing the reconfiguration capabilities of the scheme we introduce a notation for lettering chips to indicate the setting of the switches. The letter of the alphabet used indicates the control line to which the chip is connected by the control swich. The use of upper or

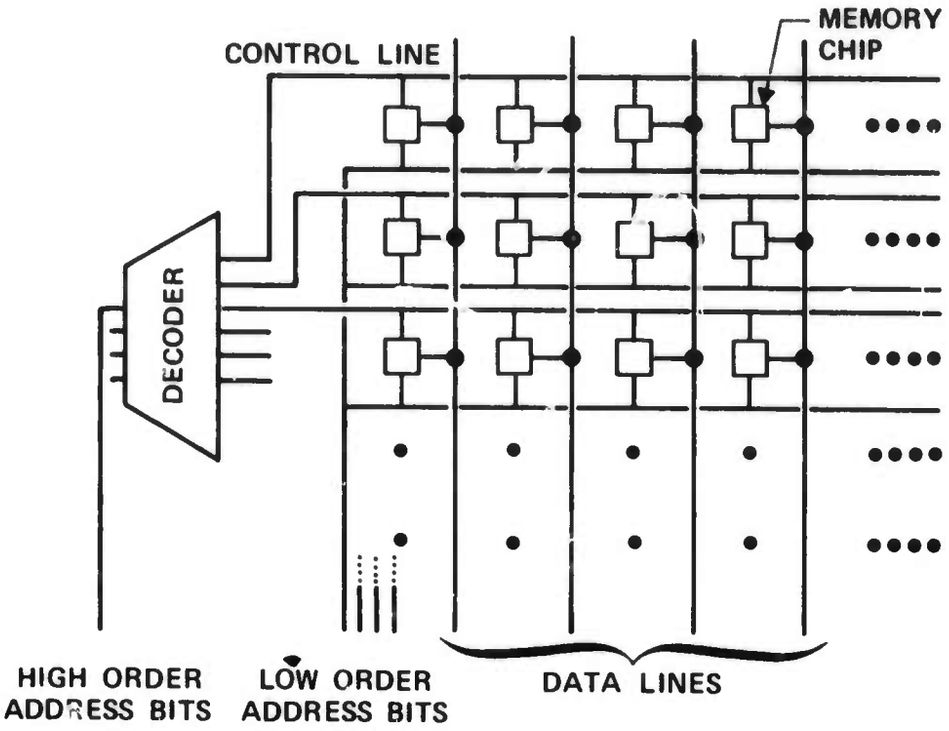


Fig. 4.1 Conventional LSI memory organization.

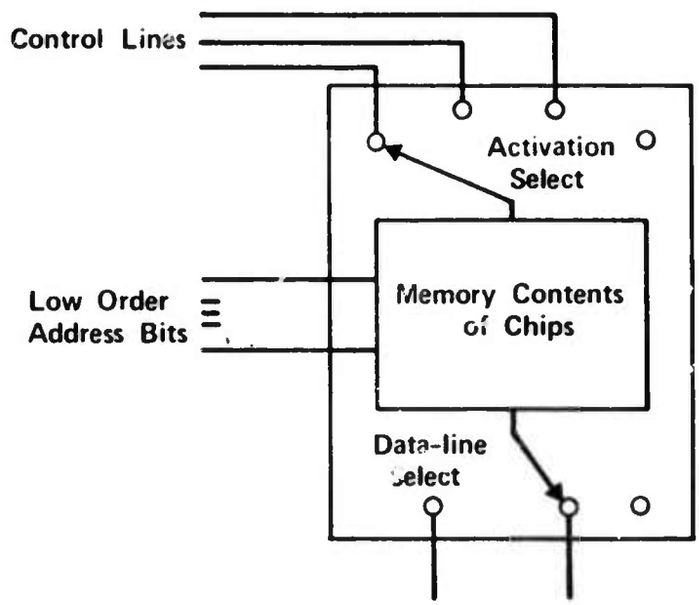


Fig. 4.2 Memory chip with components of input and output switches.

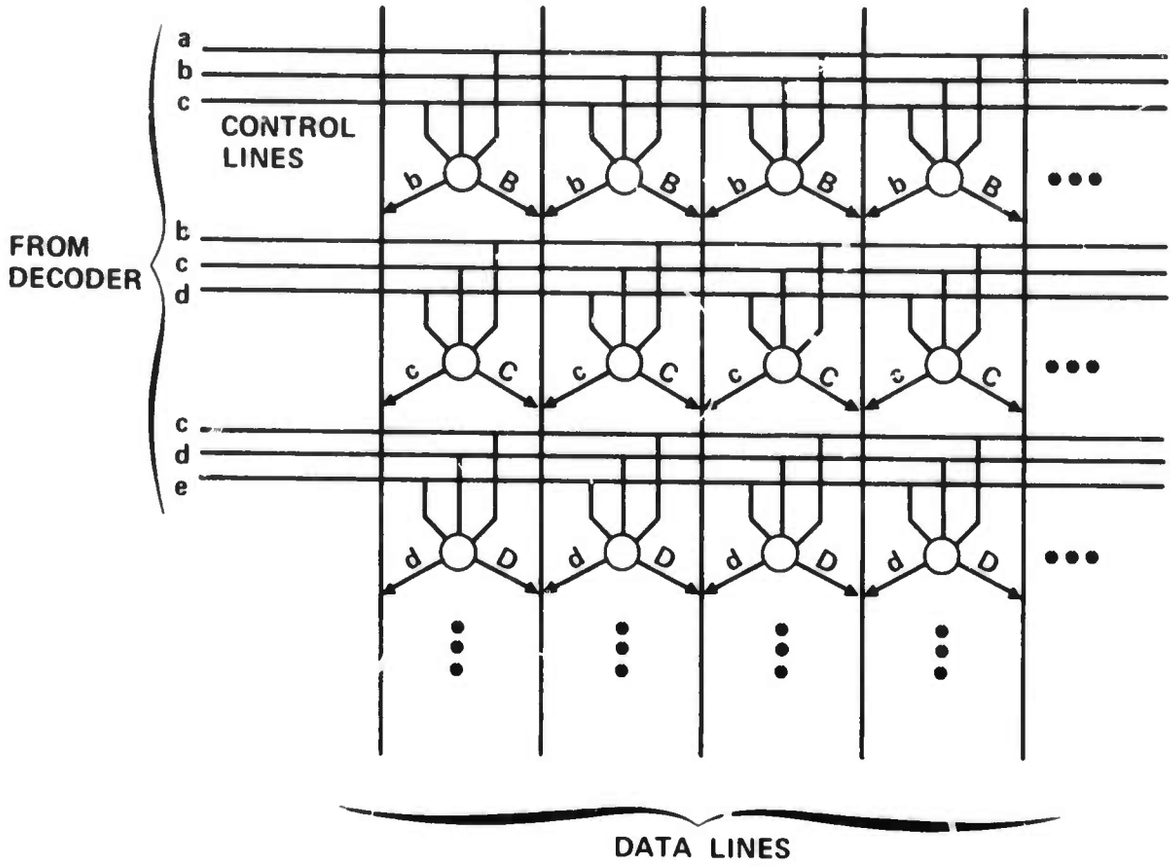


Fig. 4.3 An example chip reconfigurable memory.

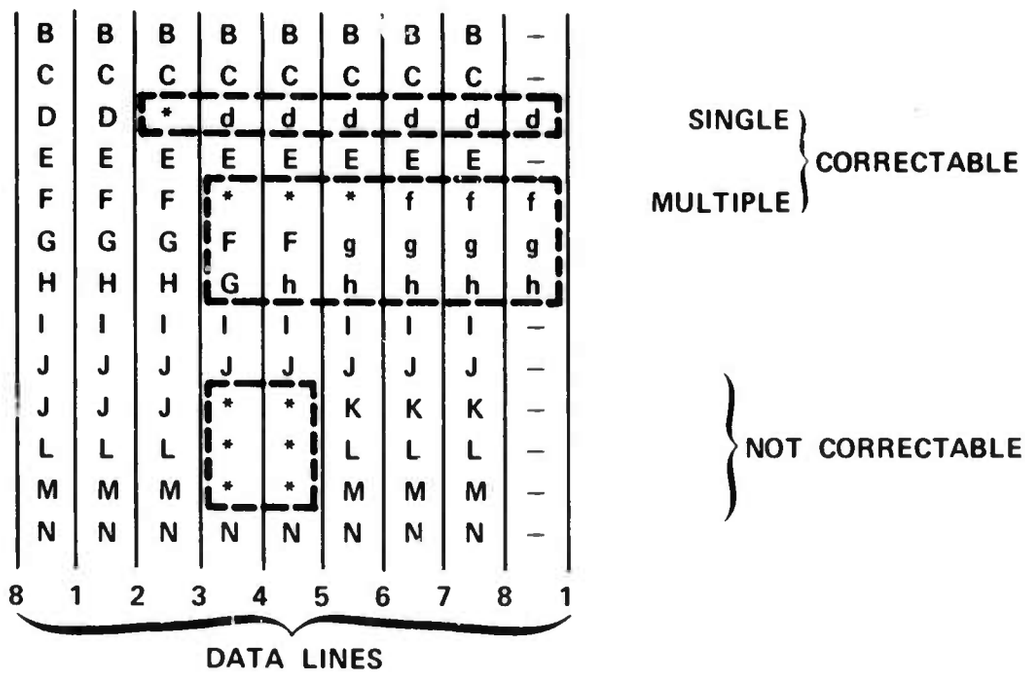


Fig. 4.4 Reconfiguration examples.

lower case letters indicates that the chip is connected to the left or right data line respectively. An unused chip is indicated by a hyphen and a faulty chip is indicated by an asterisk.

Consider the reconfiguration examples shown in Figure 4.4. The normal setting of the switches is such that the chips serve the data lines to their right. In the third row we show the case of a single faulty chip in the third column. The switches of the chips to the right of the faulty chip are changed so that they serve the data lines to their left thereby enabling all data lines to be served. In this example the reconfiguration was carried out within a single row, without having to change the setting of the control line switches in the chips. More extensive fault patterns must in general be handled by using spare chips from adjacent rows. The three faulty chips of the fifth row are handled by the following switch settings:

*Switch all good chips of row f that are on the right of the faulty chip so that they serve the data line on their left, thus replacing one of the faulty chips and leaving two vertical busses still to be served by f-driven chips.

*Use two chips of the next row (labelled F) to replace the two places in the f row that have not been handled and switch the g chips to their right to serve their left busses. This leaves vertical bus still to be served by a g-driven chip.

*Use one chip from the next row (labelled G) to handle the remaining chip position of the g row.

It can be seen that a fault pattern of n chips in a row can be handled within n rows and further that the rows below it or above may be used for the reconfiguration. In general, the pattern employed to accomodate a fault is not uniquely determined. The pattern employed may be chosen so as to better accomodate other possible nearby faults.

The third example of fault patterns shown at the bottom of Figure 4.4,

illustrates a pattern that cannot be handled by this scheme, because there is no chip that can serve row L for data line 4. This is the smallest pattern of faults that cannot be handled by the scheme. There are indeed many fault patterns containing more than 6 faults that can be handled. For example all chips in two adjacent rows can be faulty and successful reconfiguration can take place so long as there are at least twice as many rows as columns in the memory.

Appendix 3 presents detailed aspects of the memory organization, including the use of coding to detect and correct errors, the setting up of the switching network, and the relative performance of this organization, as compared with block replacement. This organization is most attractive for long-life and/or maintenance-free applications.

Beyond the number of chips required to realize a given memory size, spare chips are provided to take over the function of failed chips. The reconfiguration is achieved with a switching network that enables the number of spare chips to be potentially as low as the number of chip failures to be tolerated. As demonstrated in Appendix 3, the cost of the switching network is surprisingly small. Further, the switching network can be embedded economically within the memory chips. Thus, since typically the number of chips in the nonredundant memory is comparatively large, the redundancy required to achieve a tolerance to a significant number of faulty chips is proportionally quite low.

This type of memory organization is particularly applicable to those situations where a large main memory is required, and unattended operation is required for periods so long that many faults may be experienced. Appendix 3 discusses:

* The memory model and a reliability calculation that demonstrates the applicability of the organization.

* Types of switching networks that can realize the reconfiguration.

* A regular switching network organization that is particularly

attractive due to the ease of embedding the switching within each chip.

* Reliability estimates of the above regular scheme.

In conclusion we have determined that the switch cost for reconfiguring the chips of a memory is small when compared with the total memory cost. We have also shown that the algorithms for deciding which switches are to be set can be simple in certain cases.

DISCUSSION OF SYSTEM ASPECTS

The key aspect of the chip replacement organization is the switkhinj network that effects the reconfiguration. Appendix 3 gives realizations of such networks wherein the switch cost per memory chip is quite nominal, and whereby the switching can be embedded entirely within the memory chips. It is expected that this organization will find utility in applications with varying requirements as to long life, large memory, low or nonexistent maintenance, and low spare redundancy. For modest requirements for which only one or two =inhts neee ts be tshe'itee between maintenance operations, conventional approaches such as simple memory block replacement probably suffice. In addition, the use of low-distance error correcting and detecting codes may be desirable whenever rollback strategies are either not permitted or not feasible.

A few theoretical problems remain, the solution of which might lead to more efficient use of this organization:

* Deriving the minimal switch complexity required as a function of the memory size, number of spares, and number of faults to be tolerated.

* Deriving optimal algorithms for deciding on the appropriate settings of switches. It would be desirable to determine tradeoffs between the switching network complexity and the set-up algorithms.

Perhaps of greater practical interest are the overall system aspects of including such a memory organization within a fault-tolerant system. We

consider these issues below, with some indication of the difficulty that each aspect introduces.

FAULT DETECTION. Conventional error detecting and correcting coding techniques can be superposed on the reconfiguration. That is, the overall bit length n can include code redundancy. A decoder then checks each word on read-out from memory, in which case an immediate indication is available of the block, and possibly the byte, in error. The reconfiguration process can them remove the offensive chip and produce a new operative block of memory. The byte-error correcting codes of Section 4.1 can be used here. Also in this organization some crucial sections of memory can be given more protection by reconfiguring certain blocks to have more code redundancy than others.

SWITCHING NETWORK FAILURES. Many switch failures merely disable the chip itself and thus can be handled the same way as chip failures. Two exceptions are switch failures that produce a solid signal on a data line or that prevent a chip from being disconnected from a given control line. Such failures require the introduction of redundant data lines. Coding techniques as described above can correct for these switch failures. Also a spare data line can be provided, at slight extra cost in switching complexity. The spare line would be activated in place of a failed line, in which case the network block that receives the memory data (usually the memory data register) extracts the d good data lines from the d+1 lines directed to it.

ADDRESS DECODER FAILURES. The memory organization is clearly sensitive to failures in the decoder that drives the control lines. It is likely that some of the decoding function can be distributed among the chips, up to the availability of pins. However, for a large memory system, most of the decoder will remain external to the chips. Since the decoder consumes perhaps three to four orders of magnitude fewer parts than the rest of memory, various fault tolerance techniques can be economically applied.

SWITCH SET-UP. With one extra control line per block the switches for

the chips can be set by applying appropriate signals to the data lines and the control lines. In this mode the switches are set one at a time, a time penalty that does not appear to be excessive.

SPAN OF RECONFIGURATION. When the memory is reconfigured subsequent to a failure, a large portion of the memory may have to be reconfigured, including operative blocks. This contrasts with a block replacement scheme for which only the affected block need be reconfigured. We have not computed bounds on the number of blocks that must be reconfigured in the organization considered. However, in many systems (e.g., in a paged environment) it is possible to dump the contents of the z-1 operative blocks onto a backup, in which case the span of the reconfiguration is not a problem. This approach is not feasible in a real-time environment where long down-time (e.g., more than 10 msec.) is unacceptable.

In conclusion we feel that there are no insurmountable problems in incorporating this memory organization into a system. The cost is small in a large memory system, and may be justified by the prevalence of memory faults in such a system. The switching techniques employed in this organization are also generally applicable to homogeneous processor arrays.

4.2.4 RELIABLE SWITCHING CAPABILITY

Previous sections discuss how the memory function can be reconfigured either at the block or chip level. It remains to be shown that a switching scheme can be designed that can be fault-tolerant.

We assume the following:

* Memory blocks containing y words, each containing its own address decoder logic. Typically y will vary from .5K to 4K.

* Control units which control access to the memory blocks. Each control unit is connected to c_i memory blocks and each memory block is

connected to c_2 control units. Some regularity is assumed in the connections. (When $c_1 = c_2$, we use the single parameter c to represent both. Under these circumstances the number of memory blocks equals the number of control units, and the two units could be constructed as a single module. Possible structures for c = 3 and 4 are shown in Figures 4.5 and 4.6.)

* A data bus structure which connects to all memory blocks.

* A block address structure which connects to all control units.

* A page address structure that connects to all memory blocks.

* Control logic.

In all regular arrays of control units and memory blocks, it is assumed that all edge connections are "wrapped around" (i.e., that linear structures are mapped onto a ring, two-dimensional structures are mapped onto a toroid, and so on).

The mode of operation is explained in terms of a 'READ' from memory. Each control unit contains registers (Block Address Registers, BAR) whose contents are the block addresses of the memory blocks to which it is connected. The block address of the required word is transmitted to all control units, where a comparison is made with BARs, and if a match is found, an enable signal is transmitted to the relevant memory block. Under no fault conditions a selected memory block will receive c enable signals. The page address (i.e. the low-order bits of the address) is transmitted to all memory blocks. The selected memory block reads the selected word and places the word on the data bus. The operation is now complete.

It is assumed that each memory block is tolerant to a number of faults (e.g., one) but that a larger number of faults will cause it to be inoperative. The primary purpose of the control unit is to allow reconfiguration of the memory. This reconfiguration is achieved by

changing the contents of the BARs in the control units which are connected to the memory blocks whose addresses must be changed. A fault in a control unit could result in an error in the 'enable' signal sent to a memory block. To prevent such a fault from causing errors, a voter is used in the memory block. Note that the voter examines only the enable line. The connection of a memory block to the data bus structure can also be controlled by the multiple enable signals, thereby preventing a faulty memory block from erroneously seizing the bus structure.

Table 4.2 summarizes the fault tolerance of an example of the type of memory system described above. We assume that LSI chips will be used with 4K bits/chip. We consider an example with 32 bit words and 256k words, and we ignore the cost of the error correcting encoder/decoder circuits. We are concerned with two measures, first, the probability that a particular failure mode will occur, and second, what the effect of that failure will be.

REDUNDANCY.

* Unprotected memory	= 2048 chips
* Memory protected by byte codes	= 2560 chips (20% redundancy)
* Memory protected by byte codes plus reconfiguration	= 2624 chips (23% redundancy)

HARDCORE. The structure as outlined contains address propagation circuits. As shown, these circuits do not possess any reconfiguration capability. In this sense they represent the "hardcore" of the system.

Table 4.2 (w = 4K, Total Storage = 256K, c = 4, Word = 32 bits)

Failure Mode	Probability/Hr	External Effect
Single chip failures in a particular memory block	4×10^{-5}	None
Single chip failure in any memory block	2.6×10^{-3}	None
Two chip failures in same block before reconfiguration	$3 \times 10^{-11} \times T$ (T = time (in secs) to reconfigure)	Loss of block plus Loss of data
Control unit fault	$6{\cdot}4 \times 10^{-5}$	None
Two adjacent control unit faults (adjacent means two control units which are connected to a common memory block)	5×10^{-10}	Possible loss of block; possible loss of total memory

VIRTUAL MEMORY. The control units map virtual block addresses to physical block addresses. These units can therefore be used, with no increase in cost, to implement virtual addressing and paging, without need for any other "paging box" or its equivalent.

FAULT-TOLERANT DATA LINE STRUCTURES

A reliable memory system can be built in which the memory chips themselves can be reconfigured if a fault occurs. A further potential cause of failure of the memory is the failure of the data lines both into and from the chips. Such failures would tend to be less frequent because the amount of equipment involved is much less than in the memory function itself. Thus, for some applications it is not necessary to protect against the failure of these lines, while in more stringent applications a means must be provided to carry out some protection. The data lines may fail in two ways. First, the equipment in those lines may itself become faulty. Second, a failure of one or more of the

memory chips may cause a data line to be subjected to erroneous signals.

We can consider two prime ways in which the data lines can be protected against faults, by coding or by the use of redundant lines. In both cases equipment must be added, to carry out the encoding and encoding, or to switch the redundant lines. The probability of faults in this additional equipment may be greater than in the data lines that are to be protected, and careful analysis must be carried out to determine if such equipment is therefore justified.

CODING ON THE DATA LINES. The use of a code for single-error correction and double-error detection on the lines protects against any single data line presenting spurious data. Such a code is quite economical for all reasonable word lengths.

REDUNDANT DATA LINES. The addition of a single data line can easily be incorporated into some memory schemes such as the chip replacement scheme discussed above.

CHAPTER 5. ARITHMETIC AND LOGIC

While very low redundancy in memory produces significant improvements in system fault tolerance, arbitrary logic may require full duplication for instantaneous error detection in any one unit, and full triplication for instantaneous error correction in any one unit. Fortunately, there are several factors that may help to reduce the cost of redundancy:

(a) Detection may not be uniformly critical in time and space. For example, partial detection may suffice, detecting only certain faults, or detecting a fault within some period of time. Similarly, some faults may be more critical than others. Also, within a particular scope of computation (e.g., an instruction, a subroutine, a block, a domain within a process, or a process), detection may be required only on exit.

(b) Instantaneous correction may be unnecessary, especially when good facilities are available for recovery and retry (with or without diagnosis).

(c) Considerable flexibility arises in the use of reconfiguration of units (e.g., through changeable microcode), with tradeoffs among degrees of redundancy, performance, and functional completeness.

(d) Many systems seem to be dominated by the costs of memory. Thus, greater relative redundancy in arithmetic, logic, and control may have little impact on the overall cost of the system.

(e) Automatic retry of an instruction during which an error has been detected is both powerful and economical. Its primary requirement is that the initial operands (e.g., in registers or memory locations) should not be overwritten during instruction execution -- or at least should be recoverable from somewhere in memory.

These factors are found to some extent in existing systems, but usually in isolation rather than as part of a systematic methodology for fault tolerance.

5.1. DETECTION AND CORRECTION OF ERRORS IN ARITHMETIC

Arithmetic operations may be checked by duplication and comparison, with hardware redundancy of about 55%. Duplication detects all errors in any one unit, but fails to detect identical errors in each of the two units. The use of dual-rail logic is also possible, with hardware redundancy about 37% and 42% cited in an arithmetic-logic unit for 64-bit and 32-bit words, respectively (Carter et al. 70). However, the class of faults covered is significantly less. The use of residue codes (e.g., Avizienis 71) can be effective, with redundancy in the range between 10% and 25%. A residue code has the advantage that it is also error detecting if used in memory. For example, in a byte-organized memory with b-bit bytes, the use of the residue $2^b - 1$ detects all errors in a byte except for the error which substitutes the all-zero byte for the all-one byte, or vice versa. The cost in memory is one redundant byte. (This cost is the same for complete byte-error detection in memory, using b interlaced parity checks -- which however do not detect errors in arithmetic.) For bit-serial and byte-serial arithmetic, duplication is both cheap and effective. For byte-serial arithmetic, residue codes are also of value (e.g., Avizienis et. al. 71). A would-be problem of multiple errors resulting from a single fault can be overcome by the use of the $(2^b - 1)$'s complement of the residue. (See Avizienis 71 for the use of inverse residue codes for repeated-use faults.) For parallel arithmetic, residue codes may offer substantial cost advantages over duplication, although care must be taken in carry-look-ahead schemes to avoid unchecked multiple errors resulting from a single fault (cf. Langdon and Tang 70); otherwise duplication may again be preferable.

Byte-organized processing is advantageous for integrated circuit implementations, and is also well suited to carry-look-ahead schemes. In a byte-organized arithmetic unit with bytes of length b, multiple errors may arise in a single byte slice (e.g. on a single chip). These are detectable by residue codes with a residue at least 2^b and relatively prime to 2^b. If the all-zero/all-one substitutions are of negligible likelihood, the residue $2^b - 1$ is ideal. (If they are likely, then alternating the physical encoding for a "1" in successive

bit positions may be useful.) Note that duplication provides BYTE-ERROR LOCATION, since the error is in the lowest-order byte position in which a discrepancy exists. However this could result from a fault in either of the two units, and then either in the byte or in the carry into the erroneous byte, so that duplication is not FAULT LOCATING. Byte-error locating arithmetic codes that are not also byte-error correcting (see below) do not otherwise seem to exist (Neumann and Rao 73).

If correction of arithmetic errors is required, triplication is clearly one alternative. There is also an extensive theory of error-correcting arithmetic codes. Such codes typically require a cost roughly equivalent to duplication of the arithmetic unit (Rao 70), instead of triplication (plus voting). These codes may also be used for error detection, detecting a wide range of multiple errors at much lower decoding cost. For byte-organized arithmetic units, the recent work of Neumann and Rao is applicable, providing codes for byte-error correction in arithmetic. See Appendix 4 for an extended version of Neumann and Rao 73. (A notation gap exists between the literature on memory coding and that on arithmetic coding, which has regretfully been perpetuated.)

The suitability of such byte-correcting arithmetic codes is not uniformly clear. It depends on the particular byte sizes and word lengths, and on the type of decoding. The redundancies required for various codes are compared in Table III of Appendix 4. Included are the minimum redundancy byte-correcting codes for memory (Hong and Patel 72, see Table 4.1), denoted by "M" in the table; the AN and gAN codes with $A = (2^b - 1)p$ (denoted by "A"); bi-residue codes with arbitrary residues $2^b - 1$ and p ("R"); multi-residue codes with generalized "low-cost" residues of the form of expression (11) of Appendix 4 (denoted by "G"); and those multi-residue codes with only low-cost residues, of the form $2^b - 1$ and $2^{d_i} - 1$ (denoted by "L").

The byte-correcting arithmetic codes also provide byte-error correction when used in memory. Some of these codes have redundancy very close to the comparable byte-error correcting codes for memory. Such codes thus have potential for efficient dual use, both in memory and in arithmetic.

Advantages of such dual use are discussed by Avizienis et al. (71), particularly with respect to the residue 15 error-detecting code used in the JPL-STAR (Avizienis A2). Other codes require substantially more redundancy, in which case they are not appropriate for such dual use. Nevertheless they remain of interest for byte-organized arithmetic.

As a favorable example, consider the length k=42, with 2-bit bytes. Here 7 bits of redundancy are required for byte correction in memory, while 8 bits are needed for several forms of arithmetic byte correction (A, R, G). In particular, the radix 4 byte-correcting multi-residue arithmetic code with low-cost residues 3 and 49 = 7x7 has the remarkable property that byte-error detection in arithmetic and memory is obtained simply by taking residues module 7. Thus byte-error detection alone is cheap and fast, with correction available if desired. Other examples are cited in Appendix 4.

There is also recent work on burst-error correcting arithmetic codes (e.g., Bow 73), although that is probably of less interest here.

In general, arithmetic-error detection is highly advantageous. Error correction may be needed only rarely, especially if instruction retry is possible in the case of intermittent faults, or if alternate means are available in the case of permanant faults. Such alternate means may include, for example:

(a) Switching a spare byte slice to replace a faulty one, e.g., using the rippler of Stiffler (73). An extreme example is that of a cyclic loop of n+1 stages; when broken by a faulty stage, there are still n consecutive correct stages. However, there are problems here in switching on read-in and read-out.)

(b) Removing the faulty byte slice, with either a degradation in precision, or the use of multiple-precision operations (possibly in microcode).

(c) In a duplicate-unit environment, discarding the faulty duplicated

unit, leaving full computational capacity, but no checking capability.

Whenever the cost of arithmetic units is typically small compared with memory costs, the reliability and availability goals can freely influence the design. Nevertheless, the cost of logical (functional) duplication need not be physical duplication. For example, a fast parallel arithmetic unit may be checked by a slower byte-serial unit, with disagreement triggering an instruction interrupt. In some cases (e.g., if the result is being written into a memory much slower than the arithmetic unit), simple instruction retry may suffice. In some cases (e.g., in a pipelined environment), some rollback may be required, although this can be minimized by judicious use of registers and memory.

In summary, high availability results from a multiplicity of units, or multiple-precision modes among degraded-performance units with removed byte-slices. High reliability results from the use of error detection with retry, rollback, and reconfiguration, and with error correction possible in extreme cases. Probabilistic detection may be adequate. Periodic interspersed diagnosis provides a useful enhancement when detection is not available directly.

5.2. ERROR DETECTION IN LOGIC OPERATIONS

For logic operations, duplication is necessary for error detection in some cases, while coding does not work -- except for modulo-two linear operations (e.g., exclusive OR). Dual-rail logic (Carter et al. 72) seems valuable, with costs potentially less than duplication for error checking. In some cases, consistency checks are available. In other cases, partial detection is acceptable, at relatively low cost (cf. Carter et al. 71a). In such cases, detection is not immediate, but occurs in a probabilistic sense within a specified period of time. As in the case of arithmetic, where alternate means are available for permanent faults, various alternatives are available for logic. These include using spare byte-slices, performing (possibly micro-programmed) two-step operations on a half unit, and (in a duplicated mode)

discarding a faulty duplicate unit to run simplex.

Still another alternative is available for logic, using the arithmetic unit to perform logic operations (e.g., micro-programed) when a logic unit is not available. If the arithmetic unit is checked, it follows that the logic operations are also checked, as seen below. It is well known that all logic operations may be derived arithmetically, given for example, the bit-wise operation $x \wedge y$, e.g.:

$$x \vee y = (x+y) - (x \wedge y), \qquad (5.1)$$
$$x \oplus y = (x+y) - 2 \cdot (x \wedge y).$$

Here "$\vee$" and "$\oplus$" denote INCLUSIVE OR and EXCLUSIVE OR, respectively. The remaining operations are normal arithmetic addition, subtraction, and multiplication ("+", "-", ".", respectively). Complementation is easily obtained when ONE's or TWO's complement representations are used.

Monteiro and Rao (72) have examined a realization of logic operations using a residue-checked arithmetic unit and an AND circuit to produce checked arithmetic and logic operations. If logic operations are relatively infrequent, little performance degradation is required to perform checked logic in arithmetic. Since the AND operation $x \wedge y$ can be available as a byproduct of the arithmetic unit, e.g., when the sum is obtained as

$$z = x+y = (x \oplus y) + 2 \cdot (x \wedge y), \qquad (5.2)$$

it is possible to generate all logic operations without the extra AND of Monteiro and Rao, although it is of course desirable to augment the byproduct AND output with the correct residue check digits. For various implementations of (5.2), the incorrectness of $x \wedge y$ results in the sum $z = x+y$ being in error. If the error is detectable (e.g, via the residue check on the result), retry and reconfiguration may be initiated, as warranted. Similarly, an error in arithmetic during the formation of a logic operation other than "$\wedge$" may be detected by the arithmetic checks on the successive arithmetic operations. However,

arithmetic overflow (one bit) must be covered by the residue code. An overflow may arise temporarily during the sequence of operations (e.g., in (5.1)), but disappears in the final result.

This approach is extendable to byte-error detection and correction. However, in cases of multiple faults, it is necesssary to assure that $x \wedge y$ is correct independently of the correctness of $x+y$. For example, a pair of cancelling (but rare) errors would not be detected by the residue check on $x+y$, e.g., +1 in position i+1 of $x \oplus y$, and -1 in position i of $x \wedge y$ in (5.2).

A final word is appropriate on the impact of technology on the relevance of the schemes discussed here. On one hand, selective replication may be relatively economical. On the other hand, the trend toward increasing the number of functions per device may make the use of duplication of gates or busses less profitable if the multiple versions of a function are all on a single device. This is because there tends to be a high correlation among faults within single devices. Further limitations on some of the techniques described here will be felt because of the limitations on the number of pins available per internal function in the new technology.

CHAPTER 6. EXAMPLES OF FAULT-TOLERANT COMPUTERS

In the subsections of this chapter, we discuss fault-tolerance requirements for computers used in different applications. Our viewpoint is that the different applications have different requirements for reliability, availability, data protection, maintainability, etc., and different opportunities for the use of fault-tolerance techniques. These different requirements and opportunities result in a variety of computer architectures. In effect we see that a single 'best' fault-tolerant computer design is not possible. However, time-sharing systems possess nearly all the requirements of fault tolerance of computers in general. Therefore, we discuss them first and treat other computer types as variants.

Each subsection deals with a different application class -- general-purpose time-shared, general-purpose batch, communication, super-fast and aerospace. For each application class we discuss the most common requirements and the most appropriate architectures to satisfy these fault-tolerance requirements. Table 6.1 is a summary, in very compact form, of the material of this section. The parameters quoted (e.g., speed, memory size) are intended to be the most common without implying that examples outside the range cannot occur. The techniques that are appropriate for each application class have been discussed in detail in the foregoing chapters. Here we attempt for specific applications to evaluate some of the architectural types discussed in Section 3.3. In addition Appendix 3 contains detailed considerations in the design of fault-tolerant memory systems.

Certain properties are common to all classes of computers (or applications) of which the following are the most important from the standpoint of fault tolerance:

* Central memory frequently dominates the cost of the system, but is also the unit that is most easily and economically protected. Selective and dynamic use of coding can be very cost-effective.

Table 6.1
Application classes, their requirements and the most relevant fault tolerance techniques

MOST COMMON REQUIREMENTS	BATCH	TIME SHARED	AERO-SPACE	SUPER-FAST	COMMUNI-CATION
Data protection	H	H	VH	L	N
O.S. protection	H	H	VH	L	L
Application program protection	L	L	VH	L	-
Correctness of results	L/H	L/H	VH	L	L
Availability	L	H	VH	L	H
Recovery time (secs)	$10^3 - 10^4$	$10^2 - 10^3$	10^{-2}	$10^3 - 10^4$	$10^2 - 10^3$
Resource sharing	Y	Y	Y	N	-
Data sharing	Y	Y	Y	Y	-
Fault tolerance cost constraint	<20%	<20%	<5	<20%	<2
Maintenance	A	A	NA	A	A/NA
Memory (words)	$10^4 - 10^6$	$10^5 - 10^7$	$10^4 - 10^6$	$10^5 - 10^7$	$10^4 - 10^5$
Speed (MOPS)	$10^5 - 10^6$	$10^5 - 10^6$	$10^5 - 10^6$	$10^6 - 10^8$	$10^5 - 10^6$

MOST RELEVANT ARCHITECTURES

UNIPROCESSOR	P: AC M: EC,ED, BR				P: AC, D M: EC,
MINI-COMPUTER					P: - M: EC, BR
ARRAY				P:S M:EC,BR	
PIPELINE				P:R M:EC,BR	
MULTIPROCESSOR	P:R,AC M:EC,ED,BR	P:R,AC M:EC,ED,BR, CR	P:T,V,AC M:EC,ED, BR, CR	P:R,AC M:EC,ED, BR	P:R,V,C M:EC,ED,BR

KEY TO UPPER TABLE

H: HIGH REQUIREMENT
L: LOW "
N: NO "
V: VERY
Y: YES
A: AVAILABLE
NA: NOT "

KEY TO LOWER TABLE

PROCESSOR (P)
ARITHMETIC CODES (AC)
DUPLICATION (D)
TRIPLICATION (T)
REPLICATION (R)
VOTING (V)
SPARING (S)
COMPARISON (C)
.....
MEMORY (M)
ERROR CORR. (EC)
ERROR DET. (ED)
BLOCK REPLACEMENT (BR)
CHIP REPLACEMENT (CR)

* The arbitrary logic of central processors is the most difficult to protect but often represents a small proportion of the total cost. Thus, replication is practical for many applications. Selective replication seems practical except when all usage has uniform criticality.

* Faults in most peripheral equipment (e.g., printers, magnetic tape units, modems) are best handled by providing spares and reconfiguring.

* In most multiprogrammed systems, a vital component is the drum or other large storage device that is used for swapping. We therefore consider the effect of faults in that unit for time-sharing applications.

In view of the above common features, it is practical to consider a representative computer system and then treat other types as variants upon it (from a fault-tolerance standpoint). As such a computer, we take one of about the scale of the Multics system (Saltzer A2). It is recognized that Multics is larger than the average installation. However, it represents a suitable system on which to apply fault-tolerance techniques, because the loss of availability or of files is significant. In addition, the cost of Multics precludes the use of crude replication techniques. Because we are considering future computers, we assume that LSI techniques will be used wherever possible. Such use of LSI includes electronics associated with peripheral equipment having no stringent speed requirements, as well as memory, where we can take advantage of the regularity of structure.

In the central processors it will generally be necessary to use the faster MSI logic technology. For a system on the scale of Multics, the analysis of the use of different fault-tolerance techniques is given in detail in Section 6.1. Treating this illustrative computer as in some sense typical of computers in general, Table 6.2 illustrates the effectiveness of different techniques. The techniques are discussed in earlier sections of this report. In examining the probabilities of error, nonavailability, etc., we do not quote values less than 10^{-8}/hr.

TABLE 6-2
EVALUATION OF FAULT-TOLERANCE TECHNIQUES

Stage Reference	% of system that is redundant	Probability of Incorrect Outputs/hr. (Application Programs)	% Memory Degradation after Single Fault	% Processor Degradation after Single Fault	Mean Time to Unavailability (days)	Recovery Time for Tolerated Failures (Anplic. Prog.)	Recovery Time for Tolerated Failures (System Prog.)
1	0	.005	100	100	8.2	NA	NA
2	6.8	.002	100	100	7.6	NA	NA
3	6.8	.002	0.3	100	21	(Note 1)	(Note 1)
4	13	.002	0	100	21+	0(2)	0(2)
5	13	.002	0.3	100	21	2 msec	2 msec
6	17	.0008	0	100	19	0(2)	0(2)
7	26	10^{-7}	0	100	13	0(2)	0(2)
8	35	10^{-6}	0	100	4.2×10^3	0	0
9 (3 cpu's)	15	.0017	33	33	5×10^3 (Note3)	Note 4	10 sec
(9 cpu's)	15	.00055	11	11	$> 10^5$		3 sec
(16 cpu's)	15	.00031	7	7	$> 10^5$		< 1 sec
10 (3 cpu's)	21	.002	33	33	5.1×10^3		0
(9 cpu's)	21	.00067	11	11	$> 10^5$	Note 4	0
(16 cpu's)	21	.00038	7	7	$> 10^5$		0
11 (3 cpu's)	52	$< 4 \times 10^{-9}$	33	33	5×10^3	10 sec	10 sec
(9 cpu's)	52	$< 10^{-10}$	11	11	$> 10^5$	3 sec	3 sec
(16 cpu's)	52	$< 10^{-10}$	7	7	$> 10^5$	< 1 sec	< 1sec
12 (3 cpu's)	23	.0025	33	33	10^4	Note 4	10 sec
(9 cpu's)	23	.0008	11	11	$> 10^5$		3 sec
(16 cpu's)	23	.0005	7	7	$> 10^5$		< 1 sec
13 (3 cpu's)	20	.003	33	33	$> 10^5$	Note 4	1 sec (Note 5)
(9 cpu's)	20	.001	11	11	$> 10^5$		< .1 sec
(16 cpu's)	20	.0006	7	7	$> 10^5$		< .1 sec

1. Recovery time is dependent on time required to reload from a previous known correct state. Typically a few seconds should suffice.
2. This is the recovery time for memory failures. For detected processor failures external maintenance is required.
3. We assume that a duplexed multicomputer is unavailable when fewer than 2 complete units remain operative.
4. The application program recovery time is dependent on the time it takes the user to detect an error plus the restarting time.
5. The recovery time is dependant on the fault location. The values given here are averaged over all fault possibilities.

This represents such a low probability that the event would be expected to occur once every 10,000 years.

6.1 GENERAL-PURPOSE TIME-SHARED COMPUTERS

The most general case is that of general-purpose systems with interactive and noninteractive use. Many other systems can be considered as special cases of such systems.

REQUIREMENTS

With all expensive equipment, there is an economic need for reliability. An additional requirement is for the integrity of data. A constraint derives from the fact that a time-shared computer may frequently be used by many users. A loss of control or data may result in the effective loss of several hours work of these users -- a severe penalty. Valid control and high integrity of data are therefore vital. The manager of such a system should be prepared to pay more to protect against faults than would the manager of a strictly noninteractive (batch) system.

Many time-sharing computers are used for long-term information processing rather than short-term computing. The long-term protection of data is therefore of vital importance. This is typically achieved by recording back-up data and program files on disc or tape at regular intervals. Another aspect of the need to protect data files is the protection that must be maintained against loss of data because of the actions of other users, or an errant operating system, either possibly being caused by a hardware fault condition. We see solutions to these problems through restricting the physical address space accessible to each component of the system. In terms of the concept of levels in Section 3.2, we need to assure by suitable hardware means that low-level software (e.g., that controlling the physical allocation of resources) must be very reliable, while higher levels must be constrained to operate only in the domain allocated to them by the low level software. The protection of the operating system is therefore the most crucial

fault-tolerance requirement of the system. The Multics system is a current example of a system that recognizes the need for protection of the innermost levels of the executive against erroneous operation at outer levels. However, the protection in Multics is against software errors at outer levels and against malevolent users, not against hardware faults. The solution for hardware faults is to provide a system in which the redundancy is variable with time so that the low-level parts of the operating system can be protected without incurring redundancy for users who do not require the protection.

Consider, as a central example, a system of structure similar to Multics, initially with the following specifications:

1	Central processor
384K	Words of memory, each 32 bits
1	Unit for file storage
1	Drum or disc for swapping

We further assume that LSI circuitry is used throughout for all units except the central processor, where the faster MSI technology is used. We can estimate the chip count for an irredundant realization as follows.

Processor.....	2000 chips
Memory	3072 chips
Disc control ..	20 chips
Drum control ..	20 chips
Total ...	5112 chips

Note that the above estimates are intended only to give the order of magnitude of the system components, no greater accuracy being required (or intended). For simplicity, we assume in the following that the memory chips are organized as 1024 bytes each of 4 bits. This assumption is not critical, because other configurations of chips would yield very similar results in the reliability analysis. To a first approximation, we can assume that system error rate will be in direct proportion to the number of chips employed, and we assume a failure

probability of 10^{-6} per chip per hour. We present the various design concepts, and the techniques to be applied in a number of stages. The result of applying various fault-tolerance techniques is shown in Table 6.2, and illustrated graphically in Figures 6.1. The method of presentation is to examine a succession of stages of adding redundancy, in some cases to improve the probability of correctness, in others to improve the probability of availability, and in others to decrease the recovery time after a failure. These stages range from techniques applied to a simplex system (Section 3.3.1) to the multiprocessor concepts discussed in Sections 3.3.2 and 3.3.3.

STAGE 1: NO REDUNDANCY

In a totally unprotected non-reconfigurable mode, we can expect the reliability characteristics to be as shown in the top row of Table 6.2.

STAGE 2: ERROR DETECTION IN MEMORY

The most obvious first step in applying redundancy for fault tolerance is in the memory. The redundancy is in the form of extra bits in the words for coding as discussed in Section 4.1. At the lowest level, a single byte per word (parity byte) reduces the probability that incorrect data is able to corrupt results before being detected. We assume no mechanisms exist for reconfiguring around the fault or for recovering the lost computation.

STAGE 3: ERROR DETECTION AND BLOCK RECONFIGURATION IN MEMORY

Memory: 9 chips per block: 8 information, 1 check
384 blocks, reconfiguration around faulty blocks
3456 chips total

Processor: 2000 chips, unreplicated.

Errors in memory are detected by the error-detecting code. At the time of detection, the faulty block is immediately identified. For convenience, we assume that the number of words per block corresponds to the number of bytes per LSI memory chip, namely 1024. The state of the computation affected by the error is essentially lost unless other measures were taken earlier to establish a recovery point.

STAGE 4: ERROR CORRECTION IN MEMORY

Memory:	10 chips per block: 8 information, 2 check
	Single byte error correction within each block
	No reconfiguration around faulty blocks
	3840 chips total
Processor:	2000 chips, unreplicated.

With increased redundancy certain error correcting codes (e.g., Hamming, distance four, byte, burst) can be used which have sufficient data to enable correction of some faults and the detection of some more extensive faults. These codes enable the computer to survive in the presence of some memory faults thereby increasing the MTBF, and also reduce the probability of incorrect results. The system instantly recovers from all single faults in memory.

STAGE 5: CODING AND RECONFIGURATION IN MEMORY

Memory:	10 chips per block: 8 information, 2 check
	Single byte error correction within each block
	Immediate switchover to operative block in response to failure
	3840 chips total
Processor:	2000 chips unreplicated.

Given block replacement in memory (see Sections 4.2.1, 4.2.4 and 4.2.5), a redundancy of 20% in the memory reduces the probability of loss of data in memory to less than 10^{-8} /hr, which is negligible with respect

to other fault probabilities. The principal advantage of Stage 5 over Stage 2 is that for about 60 percent of all faults (i.e., those in memory), the recovery time is essentially zero because the combination of coding and reconfiguration allow the system to continue operation with only a small loss of memory capacity. A good strategy to follow is to transfer the contents of a faulty block to another block or to disc before another block error occurs. (In Multics this transfer is relatively easy, except when the first block of memory is affected.) Clearly, such conditions resulting from chip failures will be insignificant compared to faults due to other causes (e.g., connectors, printed circuit boards, power supplies). The number of such components will tend to be roughly proportional to the number of chips used, and the decrease, because of the use of LSI, will allow the use of more rigorous construction and testing techniques, both of which will reduce the fault probability.

STAGE 6: CODING AND RECONFIGURATION IN MEMORY,
CODING IN THE PROCESSOR

Memory: Same as 5

Processor: Unreplicated portion--800 chips
Coded portion: 1200 information chips, 240 check chips
(Assume all single chip errors are detected)
2240 chips total

As an alternative development, we may apply coding in the processor itself. Clearly there are some parts of the processor in which coding is more easily applied than in others. We estimate that 60 percent of the processor can be checked for single faults by applying coding to the following types of units:

Registers
Busses
Adders/subtracters
Counters

We further estimate that coding in the processor adds 20 percent to the chip count of the above unit types. The remaining units are mainly concerned with control rather than arithmetic. We take as a conservative estimate that any fault in the noncoded 40 percent will cause some incorrect results. Because the coding at this level is used only for detection, it does not improve the availability but does reduce the probability of incorrect results. It also shortens the recovery time in the protected portion of the system, by providing diagnostic information.

STAGE 7: CODING PLUS RECONFIGURATION IN MEMORY,
CODING OR DUPLICATION IN THE PROCESSOR

Memory: Same as 5

Processor: Duplicated portion 800 + 800 chips
Coded portion: same as 6
3040 chips total.

For further protection against the possibility of incorrect results, we take Stage 6, with the addition of duplication (and comparison) of those parts of the processor that could not be protected by coding. This addition drastically decreases the error probability, but with a slight reduction in availability.

STAGE 8: CODING AND RECONFIGURATION IN MEMORY,
ERROR CORRECTING CODES PLUS TRIPLICATION IN PROCESSORS

Memory: Same as 5

Processor: Triplicated portion: 3 × 800 chips = 2400 chips
Coded portion: 1200 information chips, 600 spare chips
(Assume all single chip errors are masked)
4200 chips total.

In this stage, the regular portion of the processor is protected by single-byte error-correcting codes. The extra cost here, including a high-sped decoder, amounts to a redundancy of 33 percent. The remaining nonregular portion of the processor is made fault tolerant by triplication. The availability, correctness, and recovery time should be adequate for practically all time-sharing installations. Thus, this stage represents the redundancy required to achive a high degree of fault tolerance in a simplex system.

STAGE 9: FIXED MULTICOMPUTER, SELECTIVE DUPLICATED REDUNDANCY

n individual computer units, n = 3 - 16, interconnected by a Communication bus

Each unit has 1.2/n the power of the simplex units in stages 1 - 8

No fault tolerance within units.

In this stage, the processing load is divided among a number of processing units. We assume that the total processing complexity is increased by 20 percent, due to the extra cost of the communication bus and due to the extra processing power needed to counteract the inefficiency of running large jobs in smaller processors. Note that the memory is also divided in a fixed manner so that large and small jobs all get essentially equal portions of main memory. The critical portion of the operating system will run simultaneously in a pair of computer units. The portion that is critical is relatively small -- comprising about 10 percent of the system overhead -- as it comprises the job dispatching and error control procedures. The recovery time after a failure in an operating system unit is dependent on the time to restart the operating system from a checkpoint. The recovery for user programs depends on the facilities available, and how they are used. Results are tabulated for decomposition into 3, 9, and 16 computers.

STAGE 10: FIXED MULTICOMPUTER, SELECTIVELY TRIPLICATED

This stage is the same as Stage 8, except that the critical portion of the operating system is run in a triplicated mode. This can reduce the recovery time after a fault occurring in the execution of the operating system.

STAGE 11: FIXED MULTICOMPUTER, UNIFORM REDUNDANCY

Here all programs are run simultaneously in two computer units. Thus the system is more than 50 percent redundant, with good availability. The recovery time is uniformly low for all programs. This stage is not of primary interest here, but is of use in aerospace environments.

STAGE 12: RECONFIGURABLE MULTICOMPUTER OR MULTIPROCESSOR, DYNAMICALLY USED DUPLICATION FOR THE OPERATING SYSTEM

Conventional multiprocessor containing n processors and n memories, n = 3 - 16

Each processor is 1.25/n the power of the simplex processor

No fault tolerance within units

Pair of processor/memory combinations can be operated in duplicated mode for error detection.

Here a processor/memory combination can be configured out of operative processors and memory blocks. It is also possible for application programs to get variable blocks of memory by appropriately configuring the switch. The switch here is more complex than the communication bus of Stages 9, 10 and 11, so that we assume the extra processing complexity required is about 25 percent as compared with the simplex processor. The critical portion of the operating system again runs in two computer units. The critical portion is larger here than in Stage 9, because the protection mechanism is more sophisticated, and must be fault tolerant. Consequently, we assume that about 40K words of main memory, and about 30 percent of the processing load, are required for the critical portion of the oprating system. The primary advantage of this stage over Stage 9 is in its increased availability, because of the partitioning into separate processor and memory units. The results for

this stage represent the performance expected for the architecture of Section 3.3.3.

STAGE 13: MULTIPROCESSOR WITH DYNAMICALLY USED DUPLICATION, FLEXIBLE INTERPROCESSOR COMMUNICATION, SELECTIVE MEMORY CODING

Conventional multiprocessor containing n processors and m memories, n, m = 3 - 16, n not necessarily equal to m
Each processor is 1.3/n the power of the simplex processor
Dynamically modifiable byte error correction within memory units
No fault tolerance within processor units
Pair of processors can be operated in a duplicated mode for error detection.

This stage differs from Stage 12 in that the switch can interconnect among processors, as well as between processors and memories. This added flexibility permits two processors to operate in a duplicated mode, without requiring the cost of memory duplication. The memories can use coding selectively, as in the case of the processors, only for the critical portion of the operating system. Because of the switch complexity, we assume that the extra processing power required for this stage is 30 percent of the simplex processor. The net effect is to increase the availability as compared with Stage 12.

6.2 GENERAL-PURPOSE BATCH PROCESSORS

In general-purpose batch applications, we include both scientifically oriented applications and those concerned with more commercially oriented tasks. The fault-tolerance requirements of both differ slightly from time-shared systems. Principal among these differences are those stemming from the need to meet deadlines, and the extreme importance in certain cases of the need to protect against the possibility of erroneous output. Techniques that are employed with success at present include the use of accounting checks in commercial operations to detect errors, and the use of checkpoint-restart to prevent excessive lost processing in the event of machine breakdown.

The most difficult criterion to meet is that on the cost of fault-tolerance hardware. Rarely will a figure in excess of 20 percent be justified for the cost of such hardware. This figure explains to a certain extent why such hardware has been restricted in existing systems, frequently being limited to parity in memory and on data transfers. However, recent computers have extended the protection to single-error correcting codes in memory (e.g., IBM 370, Burroughs 7700), and even to the use of residue codes (see Section 5.1) in the arithmetic unit of the Burroughs 7700. Reconfiguration in the event of faults has also been introduced, in such units as memory blocks, I/O channels, power supplies and peripheral equipment.

The trend of decreasing cost of electronics (compared with other costs such as manpower) will continue, and also the use of such computers for more and more time-critical calculations. We can therefore expect to see a move toward computers with a greater demand for fault-tolerance than at present.

The architectures most suited to general purpose batch operations will probably employ fault-tolerance measures that are relatively close to those used in time-shared computers. The one area in which significant differences will be found is in the peripheral equipment so essential to batch-operated computers, particularly those used for commercial EDP operations. In installations that require high reliability, present practice is to use a large number of each type of peripheral so that the loss of one unit causes only a small decrease in the throughput capabilities. This practice is already successful in providing the necessary high availabilty.

6.3. COMMUNICATIONS PROCESSORS

In communications processors, we are concerned with processors for three main functions:

* Message switching, e.g., the Bell system ESS

* Message store and forward, e.g., the interface message processor (IMP)
* Front-end processing, e.g., the terminal interface processor (TIP).

These functions are so closely related that a combination of them often coexists in one computer.

REQUIREMENTS

A communications processor is always part of a much larger system. The important requirement is reliability of the system as a whole, and we therefore expect that efforts would be made to design the system so that faulty processors do not interrupt service within the system. A fault in a processor that was acting as a front-end processor or as a connecting point for one of the host computers of the system would isolate either users or some facilities from the system but should not cause serious degradation of the remainder of the system. Such a front-end processor should be at least as reliable as the host computer attached to it. Certainly, one order of magnitude is sufficient for the improved reliability over the host, and more stringent requirements are unrealistic.

Another potential reliability requirement is for the protection of data. In most communications systems, data protection is not of significance in the individual communications processors but should be achieved at the system level, e.g., using such techniques as coding applied to the messages to detect errors, and retransmission by alternative routes to achieve error recovery. This system emphasis has implications on how recovery from faults can be achieved, in that it is not necessary to remember the state of the processor at the time of the fault in order to restart it after appropriate corrective action has taken place.

Because retransmission can be used to accomplish recovery from a faulty message, it is far more important to design the processors and other components of the system to achieve error detection than to provide error-correction capabilities.

Communications processors are often located at sites with no resident maintenance staff. Therefore, the diagnosis and repair of faults should (if possible) be carried out from distant points in the network. Diagnosis of some fault conditions is possible, but many other conditions render the faulty processor incapable of communicating anything meaningful to the other parts of the communication network. Therefore these other conditions present a significant problem in diagnosis.

A RELIABLE IMP

To illustrate the design concepts appropriate for a communications processor, consider the IMP currently used on the ARPANET. This processor carries out all of the functions mentioned above (switching, store and forward, terminal and host-computer interfacing). The IMP uses a Honeywell 516 with additional electronics principally to interface to the communication equipment and host computers.

As an approximation the H516 contains about 1600 ICs, each of which contains, (on average) about 10 gates. Assuming that chip failures are a significant proportion of total hardware failures, and assuming a failure rate of 10^{-6} per chip per hour, we can expect a failure rate of 0.0016 per H516 per hour, or about 13 per H516 per year.

As of August 1972 (see BBN's "Network Summary", Aug 1972), 31.6 IMP years had been logged. With the above assumption, we would expect about 400 chip failures. The number of unscheduled down times over this period was 881. In resolving these figures (400 and 881), we point out that:

* The 881 includes software and external power-supply failures.
* The number 400 excludes many other failures, e.g., of the core memory, passive components, and connectors.
* Marginal conditions corrected during preventive maintenance are not included in the 881 unscheduled down times.

As a conclusion, we regard 10^{-6} failures per chip per hour as a realistic but perhaps conservative estimate of chip failure probability.

We now consider the design of a more reliable processor for the IMP environment. We consider two possible realizations, MSI--Medium scale integration (about 100 gates/chip) with a core memory, and LSI--Large scale integration with a semiconductor memory. We reject the possiblity of using small scale integration (SSI) in any future development.

We can expect that with even an MSI realization, the number of chips required will be reduced by a ratio of about 10:1 to approximately 160 chips with an attendant improvement in reliability. In addition the number of connectors will also be reduced. We can expect that failures because of active components will be reduced to about 1.5 per year per IMP. In an LSI representation the memory would require about 128 chips (assuming 4K bits per chip and 32K words of 16 bits), and the processor about 16 chips, resulting in approximately the same (1.5) number of faults per year in the active circuits.

Against the above projected failure rates, we must compare failures due to non-electronic causes, e.g., city power failures. These latter failures will dominate. It is therefore our conclusion that the correct design policy is:

* Use MSI or LSI circuitry whenever possible.

* Maintain message integrity on a system basis.

* Maintain system integrity by re-routing on a system basis.

* Improve overall reliability, e.g., by improving the reliability of software, or that of power service.

We further point out that a failure rate of 1.5 per year for an IMP-like processor is expected to be far better than most of the host computers to which they are attached.

As corroboration of the above viewpoint, we note that the communication processors used by Tymshare Inc in their TYMSAT system experience an average of 1.5 failures per year. There are 93 such processors in use. The processors are Varian 620 computers which is comparable in capability with the Honeywell 516 used in the IMP.

There is a choice of how much fault tolerance to put in the IMPs. Some investment in IMP reliability is worthwhile in light of the expected increase in the availability of hosts via the IMPs.

A RELIABLE HIGH-PERFORMANCE COMMUNICATIONS PROCESSOR

As seen above, a reasonably reliable IMP can be built without resorting to any special fault-tolerance techniques. This possibility ceases to exist if a communication processor is to be designed for significantly higher performance.

For the purposes of this subsection, we consider a high-performance communications processor that contains an order of magnitude more components than the IMP replacement discussed above. Assuming the same technology, this greater complexiity would increase the expected number of failures per year from 1.5 to 15, an unacceptable increase which must be handled by the use of fault-tolerance techniques.

In addition, we can envisage an increase in traffic on the network. At present on the ARPANET, the traffic load is small enough that the rerouting of messages can be used as a technique to prevent a faulty IMP from affecting other parts of the system. As the traffic load increases, this technique becomes less viable, and it becomes necessary for the communication processors to be more reliable.

We are concerned with three aspects -- error detection, error correction, and processor availability. It is recommended that error detection be carried out by coding on the messages (or packets). The overhead associated with the coding bits is very small for packets of

the order of a thousand bits. Because of the tendency of line failures to produce bursts of errors, a burst detection code should be used.

Retransmission represents the most satisfactory technique for error correction, and where possible, this retransmission should be over an alternative route. However, it is necessary in a heavily loaded system that the number of such retransmissions be kept acceptably low. This constraint implies a requirement for rapid detection, diagnosis, and correction of any fault condition in any of the processors.

The dual requirements of high performance and high availability suggest a multiprocessor or multicomputer approach. Two such systems exist in the design state: the projected new high-performance IMP design (Ornstein A2, and Heart 73), and the PLESSEY 250 (Williams A2). In both cases, high availability of the processor is to be achieved by switching out faulty processors and using other units to take over the workload. A problem that can occur in such schemes is that the unit carrying out the disconnection must itself be very reliable so that one can guarantee that a faulty unit cannot corrupt the whole system, i.e., we need to achieve a significant degree of fault isolation.

In the case of the new IMP design, the disconnection is carried out by sending a code word to the bus interface of the bad processor. The use of a code word rather than a single control signal is intended to prevent other processors that are faulty from turning off good processes. The use of a code word, and therefore the need to recognize the correct code word, will increase the complexity of the logic that carries out the disconnection, thereby tending to make that logic less reliable. On the other hand, there is a somewhat better probability that a bad processor will not accidentally turn off good processors. Other schemes to carry out this operation have been investigated and appear to have some merit. Principal among such schemes is one used in the PRIME system at Berkeley (Borgersen A2). In that scheme, if a processor decides to turn off another processor, it asks a third processor to carry out this function. The third processor validates the operation before carrying it out. In general, therefore, two processors

must be at fault before incorrect disconnection is carried out. This rule is not entirely true for all possible fault conditions -- for example, if the third processor was faulty and erroneously believed that it had been told to turn off the second processor, then incorrect disconnection would occur, caused by a fault in only one processor. An improved scheme for carrying out disconnection could use the combined logic of several processors to turn off any other processor. For example, to turn off processor 4 would require processors 1, 2, and 3 to disconnect it. To turn off 5, the logic of 2, 3 and 4 would be used, and so on. The disconnect function would include a voter from the three control signals, thereby preventing any single processor at fault from being able to turn off any other. Such a scheme would prevent a single faulty processor causing erroneous disconnection. Yet, because the voter contains significantly less logic than the code recognizer of the IMP scheme, the immproved scheme would achieve greater system reliability. Schemes such as those discussed above are all possible in the PLESSEY 250 system, where such actions are carried out by program.

Even if the new IMP design achieves 100 percent availability, it will still suffer from many of the breakdown situations that occur with the present IMP. Significant among these are software bugs, the breakdown of lines between IMPs, the loss of power to the computer, and occasional catastrophes such as when the IMP at Lincoln Lab was affected by a lightning strike. The operation of a very reliable network must be carried out with significant management attention to such matters. In the case of modern LSI machines, with their potentially low power drain, it is entirely practical to use standby power supplies. In addition, the processors can be placed in a protected environment to avoid problems due to temperature extremes or other environmental conditions. Software troubles can be removed primarily by increased validation of programs before their use. Such validation at present cannot be carried out fully because the lack of a sufficiently large test facility at Bolt Beranek and Newman. In the more general case of communication processes for other than the research community (such as the present ARPA network), we can envisage a much more stable operating environment with fewer program changes. In that case, the software troubles should be

significantly reduced. Stability of operating environment is certainly the case with more established networks -- for example, Tymshare's network, where software troubles are negligible.

In examining the ARPA network, we see ways in which the network can achieve higher availability by the use of a few replicated lines. The computers on the network are mostly in fairly tight geographical clusters, and few IMPs are heavily loaded. We can therefore envisage the multiple connection of certain computers to IMPs as "very distant hosts". A particular grouping could, for example, be SRI, Stanford University, NASA Ames, and Berkeley, which could be multiply connected to each other's IMPs. Another such grouping could include MIT, Harvard, Lincoln Lab and BBN. These connections could be accomplished in such a way that, if an IMP were lost, the hosts attached to that IMP would operate as very distant hosts of the other IMPs. This hookup would prevent the hosts from losing their connection to the network. This technique is not 100 percent useful, as some computers (e.g., University of Utah) are not geographically close to other IMPs. However, the total system reliability could be significantly improved at low cost.

6.4. SUPER-FAST COMPUTERS

Several super-fast computers exist or are in development. Notable examples are the CDC STAR, the Texas Instrument ASC, the ILLIAC 4, and the Goodyear STARAN. The structure of these computers differs substantially from conventional computers. In this subsection, we examine fault-tolerance techniques that are appropriate to this class of computers.

REQUIREMENTS

Such computers frequently cost significantly in excess of 10 million dollars. Backup alternative computers seldom exist, principally because few models of each computer are produced, and in certain cases there is only one in existence. Some of the applications for these computers have a demand for high reliability. An example of this case is the

control of a ballistic missile defense system.

The great complexity of these computers plus the very high speed of their circuitry tends to make fault diagnosis a very complex process. The great complexity also increases the component count, each added component increasing the unreliability of the system, thereby tending to make the MTBF much worse. In the case of ILLIAC 4, the MTBF is currently approximately five hours.

TECHNIQUES

The large memories associated with the super computers tend to enhance the system benefit of memory fault-tolerance techniques. Typically, the memory will be a very high portion of the total component count within the system. The techniques of coding and reconfiguration as discussed in Chapter 4 and Appendix 3 are applicable to such systems, and for a redundancy of the order of 25 percent can provide highly reliable memories whereas by the use of coding alone, the lower redundancy will still produce acceptably good reliability. The only drawback to the use of such techniques in these machines is the fact that they add a certain number of gate delays in the access time to the memories, whereas such computers generally are designed to operate the memory as fast as possible. The use of look-aside pipeline decoding (Carter et al. 72b) prevents the decoding delay from having a serious impact on the processing speed.

In the case of pipelined arithmetic units such as the ASC and the STAR computers, the pipelining of arithmetic checking by residue codes or other means can be carried out in parallel with the main processing pipe. The result of checking in parallel is a very small delay to the arithmetic operations, since the syndrome generation adds only 2 gate delays to the length of the pipe.

Large computers are frequently used on calculations, the correctness of which can be verified by what we could call algorithmic checking. This is the carrying out of a subsidiary calculation that will form a check

as to the correctness of the first calculation. An obvious example is to reinvert a matrix after the original inversion process to see if the resulting matrix is the same as the original (within certain bounds, to allow for round off error). Many examples of this type of checking exist. We cite a few below.

PARTIAL DIFFERENTIAL EQUATIONS

Many partial differential equations can be checked by examining the validity of the governing equation at each point in the mesh. For certain equations, this represents a task equal in size to the original solution of the equations. However, for some partial differential equations, such as boundary value problems, the checking for correct solution is significantly easier than finding the solution.

MATRIX OPERATIONS

The reinversion of the matrix as mentioned above provides a check. However, this essentially doubles the total work performed. We can instead carry out a related calculation -- for example, we can multiply the inverse by an arbitrary vector x yielding a vector y. By also multiplying the original vector matrix by y, we should obtain x, the original arbitrary vector. Although this method is not 100 percent certain most faults in a computer will be detected in this manner.

The calculation of eigenvectors and eigenvalues can be checked by the fundamental relationship that

$$A \underline{x} = \lambda \underline{x}$$

In addition, in certain matrix calculations a check sum or several check sums can be carried on along with the calculation. In most methods for inverting matrices, it is typical to compute a row sum at each state of the pivotal condensation method. The row sums provide a check for the remaining calculation.

In summary, where a regularity in a mathematical sense exists in the

calculation, a simple inverse calculation can often be performed which provides some capability for checking the original calculation.

In the case of array computers such as the Illiac 4 and the STARAN computers, algorithmic checks as discussed above can be carried out by adding extra processors that perform this checking at all times in the calculation. This would result in a certain redundancy of equipment, but would speed up the checking process.

It must be pointed out that the techniques discussed above are not universally applicable. The major reasons that preclude their use are lack of storage to retain partial results, lack of bandwidth to place partial results on a back-up memory, and the lack of an efficient inverse calculation. These considerations make it necessary in such cases to use other fault-tolerance techniques.

Reconfiguration in array computers is complicated by the fact that the communication paths between each processing element and its neighbors are of very high bandwidth and contain a large number of lines. Therefore, if a substitute processor is to be inserted, the switching capability has to be very large. In addition, extra gate delays that may be introduced by such switching capability will frequently not be tolerable. In the case of these machines, manual switching of a new processing element into a mesh of such elements appears the most practical form for rapid reconfiguration in the event of faults in a processing element.

Beyond the special points mentioned above, reliability techniques for array processors are effectively similar to those for any other type of computer. They are conditioned by the fact that, since the processors are so large, the probability of failure is much higher. The switching problem in reconfiguration is thus complicated by the large number of lines of high bandwidth. However, these disadvantages are of less critical importance, because such machines are seldom used in a real-time on-line mode, and a few minutes downtime for manual reconfiguration is often acceptable.

Super-fast computers do present one problem not always apparent in other systems, namely, the difficulty of checkpointing the total state of the system so that the computer can be switched back to that state at some future time for a calculation to be restarted. Checkpointing can be a complex and time consuming operation in a computer that is very large and in which many operations are carried out simultaneously.

6.5. AEROSPACE COMPUTERS

Aerospace computer systems have been considered in great detail elsewhere (e.g., see several systems in Appendix 2). They are treated here partly for historical reasons, partly because experience in such systems is relevant to parts of more general systems, and partly because their redundancy can be reduced in many cases.

REQUIREMENTS

Aerospace computers differ from many other computers in several ways. We discuss here the differences in the requirements for fault-tolerance. For example, we can express one difference in terms of a requirement that the probability of an incorrect result being generated should be less than 1 in 100 million per hour of use. This is the relevant figure for calculations critical to flight safety in a commercial aircraft (Wensley et al. 73). It translates into a MTBF of 10,000 years, a very stringent requirement upon reliability.

In addition, for certain calculations such as stability augmentation or flutter control, the recovery time must be exceedingly low. In certain cases, it is as little as 10 milliseconds.

In this application field the computer is a very small proportion of the total cost of a system, whether a commercial jet liner or a space vehicle. Thus, a level of redundancy may be afforded that in many other applications is not economically practical.

It is typical in the aerospace field that highly repetitive calculations must be performed, and that these are very loosely coupled. Such calculations typically might compute the numerical solution of differential equations that represent a mathematical analog of a control servo. The iterative nature of the calculation can be taken advantage of by carrying out the checks at the end of each iteration, rather than at the end of every small operation within the iteration.

In certain aerospace applications, no maintenance is available, particularly in the long-life space missions to the outer planets. The fault-tolerance procedures must be automatic because in addition to the lack of maintenance availability, there may be occasions when such a space vehicle could be in a position where communication with the earth was either not possible or of very low bandwidth. In addition the life expected from computers in such missions may be very high, from five to ten years being entirely possible. Such long life means that the probabilities of chip failures and other malfunctions become very high, to the point that over half of the circuits within the computer may have failed.

TECHNIQUES

The most obvious technique to use for both detection and correction is extensive replication, usually triplication, Also, the application is well suited to a multiprocessor organization that can handle many independent processes. The output of several identical processing elements is compared and voters attempt to remove the effect of one of the processors being in error. Although coding is also used to assist in error detection and correction, coding alone is not sufficient to provide adequate reliability for some of the most critical applications. As mentioned above, voting may be carried out at the end of each iteration of a repetitive task, as in the SIFT system (Wensley A2, 72), or it may be carried out upon each transfer of data between processor and memory, as in the Hopkins system (Hopkins A2) or in the BUCS system (Wensley et al. 73).

The reconfiguration of a system is typically accomplished by switching out faulty processors and switching in some spare processors or memory modules. In the case of the Hopkins scheme, a multiplicity of units is switched in and out whenever a fault is detected. Two processes and three scratchpad memories are all discarded and the calculation is transferred to another module of the same size.

While the degree of redundancy that is acceptable and needed in aerospace applications is very seldom appropriate to large ground based systems, the techniques may be very usefully applied to certain small critical subsystems within a large system.

6.6. CONCLUSIONS

The main conclusions to be drawn from our study of applications and architectures for fault tolerance are:

(a) Many existing computer designs already incorporate some fault-tolerance techniques which in some application fields provide adequate availability and guarantees of correctness. Prime examples are those systems used in financial institutions (banks, stock exchanges etc.,) and commercially operated service bureaus, with both batch and time-shared modes of operation.

(b) Computers that are built using the newer technologies (e.g., LSI) are intrinsically more reliable, primarily because of the reduced number of components and the attendant reduction in the number of such items as connectors and cables.

(c) Techniques exist to provide adequate fault-tolerance for all application fields. In most cases, these techniques are economical, especially when compared to total system costs.

(d) Different techniques are sometimes necessary for improvement of

different fault-tolerance parameters, e.g., correctness, availability or recovery. The proper specification of fault-tolerance must recognise these different parameters.

(e) The use of selective redundancy can be an effective technique to provide greater fault-tolerance for critical system functions and smaller redundancy for non-critical programs.

CHAPTER 7. CONCLUSIONS AND RECOMMENDATIONS

7.1. CONCLUSIONS

This section summarizes the main conclusions of the report.

GENERAL CONCLUSIONS (See Chapters 1, 3 and 6)

* Techniques exist for achieving economical fault tolerance for many important applications, without needing massive redundancy. Significant levels of correctness and system availability can be achieved with redundancy from 10 to 40 percent.

* Techniques exist to provide a much higher degree of graceful degradation than is currently available.

* A significant problem in existing systems is the unpredictable and unnecessarily long time required to recover after the occurrence of some faults. This problem is made worse in most existing systems by poor architectural structures and inadequate diagnostic techniques.

* The degree of fault tolerance required and the choice of techniques needed to achieve it are both strongly dependent on the environment.

* Software and operational considerations must be carefully integrated with the hardware in the design of a fault-tolerant system. The present art of computer system design is capable of such integration, if properly motivated by managment directives.

The following discussion concerns some of the specific techniques for fault tolerance. Some of these are readily available, while others are capable of being developed.

ARCHITECTURAL CONSIDERATIONS (See Sections 3.2, 3.3, Chapter 6)

* Simplex systems are adequate in some cases.

* Reconfigurable multiprocessors are desirable for high availability and graceful degradation.

* Good system structuring is highly beneficial throughout system development.

* System security is strongly related to fault tolerance. Protection mechanisms are critical to some uses of multiprocessor architectures.

PROCESSOR CONSIDERATIONS (See Chapters 3, 5 and 6)

* In most systems, dynamically selective replication of critical processing capability may be used without greatly affecting the overall cost.

* Deferred detection, interspersed on-line diagnostics, and automatic recovery strategies are useful in reducing redundancy when time is not critical.

* Error detection (or correction) in arithmetic can be achieved with codes also achieving error detection (or correction) in memory (see below), at almost the same cost as the best codes for memory alone. Byte coding is suitable for LSI arithmetic.

* For certain processing functions, increased dependence on memory (e.g., by table driving) is very effective, since it allows economical use of redundancy. Distributed logic-in-memory designs are interesting in certain cases.

* The use of read-only memories with coding can be highly effective for reliable logic.

MEMORY CONSIDERATIONS (See Section 3.1 and Chapter 4)

* Fault tolerance is more economical in memory units than in other parts

of a computer system. Performing functions in memory that are normally done in logic (e.g., via table-driving) permits economical fault tolerance.

* Coding, byte slicing, page relocation, and memory reconfiguration are appropriate for fault-tolerant memories.

* Byte slicing and byte coding are particularly appropriate for LSI memories that store several bit positions on a chip. Byte coding requires only one redundant byte per word for detection of arbitrary errors within any byte of the word, and a logarithmically increasing cost for byte error correction. The increase in the overall cost due to encoding and decoding is negligible (except for very small memories).

* No delay is required for decoding in the absence of errors whenever error detection (syndrome generation) can be overlapped with execution in an automatic instruction retry environment.

* Reconfiguration around faulty memory components is simple and highly effective. Reconfiguration at the block level is aided by page relocation in hardware. A virtual memory organization in hardware can offer further benefits for fault tolerance. For certain high-availability and high-reliability requirements, replacement by switching at the chip level is appropriate in combination with byte coding.

TECHNOLOGICAL CONSIDERATIONS (See Chapters 3,4,5,6)

* Newer technologies permit certain techniques for fault tolerance to be practical. However they do not supplant the need for architectural fault tolerance.

* LSI outmodes many of the techniques for handling single faults and single-bit errors. Correlated faults must be considered.

7.2. RECOMMENDATIONS FOR FUTURE RESEARCH AND DEVELOPMENT

Throughout this report are conclusions with implications for future research and development. Our recommendations for future research and development are summarized here, and are classified according to detection and diagnosis, architecture, and analysis.

DETECTION AND DIAGNOSIS

Most existing fault-tolerant systems use primitive techniques for error detection (e.g., replication of processors, coding within memory). We remain convinced that more economical methods exist, such as using probabilistic and deferred error detection, which, for example, take advantage of knowledge about existing permanent faults. Feedback error detection is also possible. Models are needed that permit a theoretical study of the time-space trade-offs in fault-tolerant systems.

Programmed consistency checks are a powerful error-detection technique for certain types of computations -- notably those involving servo-type control or those with a readily computed inverse. We believe that a much broader class of programs is suited to such checks. The use of run-time assertions (e.g., similar to in nature, but not as complete as, Floyd assertions) appears to be very promising.

Periodic self-diagnosis is important as a means for fault detection and also as a means for reducing needs for preventive maintenance and eliminating the need for emergency maintenance. Good algorithms now exist for specifying test sequences for combinational networks when the faults are simple, e.g., gate outputs being stuck at 0 or 1, but not for more realistic faults. The sequential case is not at all well understood. Very little has been done on the problem of general methods for diagnosing large systems so as to pinpoint a faulty module. We feel that these problems are all soluble if specific structures (say, distributed two-dimensional networks) are considered, or if redundancy is permitted within the logic to enhance diagnosability.

ARCHITECTURE

A serious weakness in the current art is the absence of a design methodology that integrates hardware and software into a systems concept addressing reliability, availability, security, efficiency, and functional capability in a unified way. For example, significant benefits can be expected from techniques for structured design and implementation (see Section 3.2). Such a methodology requires significant communication and cooperation among research and development people, among hardware and software people, and among university and industrial people. (The ARPA Network is providing some steps in this direction.)

There is a need to develop economical architectures for fault tolerance in a general-purpose environment. (The aerospace and telecommunications applications and specialized minicomputers have received most of the attention to date.) In particular, the multiprocessor outlined in Section 3.3.3 is an attractive possibility, with selective replication in time and space. An operating system for this architecture is also worth investigation. There is also a need for an economical solution to the protection problem in a large dependent-processor multiprocessor system.

Possibilities for fault tolerance should also be exploited via novel architectures, including highly reconfigurable distributed micro-processor arrays and networks of larger computers. An important direction for future systems is the achievement of smoothly degradable economical systems with rapid recovery from faults. The scheme for reconfigurable memory arrays of Section 4.2 represents a possible starting point for such systems.

ANALYSIS

There remains a difficult problem of analyzing the reliability of a redundant system or even proving that it is, say, single-fault tolerant. The difficulty is greatly reduced by structured design and by proofs

that the executive can reconfigure the system as intended. This issue is no different from proving the correctness of an operating system -- a process considerably simplified by structured design. However, the modeling of complex fault-tolerant systems is also important here -- an issue frequently studied, but still not adequately resolved.

An important quantitative measure of a fault-tolerant system is the relative cost of fault tolerance, e.g., the redundancy. Except when trivial techniques are used, it is difficult to estimate the redundancy accurately. In this report we associate the various redundancy techniques with different types of architecture. More generally, it would be useful to have measures of the total redundancy, e.g., as a function of availability, reliability, and down-time.

In summary, the state of the art leads to considerable hope for the development of economical fault-tolerant systems. However, there is still much need -- and fortunately, much room -- for advancement in the state of the art.

CHAPTER 8. REFERENCES

The main references cited in the text are included here. Other references are found in the appendices. Several extensive bibliographies are worthy of special mention, and are listed first.

BIBLIOGRAPHIES

* B. D. Carroll and E. W. Smith, A Bibliography of Fault Tolerant Computing, Auburn University Technical Report AU-T-22, February 1972 (for Army Missile Command), AD 739 522. Includes 422 entries.

* P. Scola, An annotated bibliography of testing and diagnostics, HONEYWELL COMPUTER JOURNAL 6, 2, pp. 97-102, 1972 (with accompanying microfiche). Includes 1300 entries, annotated.

* R. A. Short, The attainment of reliable digital systems through the use of redundancy -- a survey, IEEE COMPUTER GROUP NEWS, pp. 2-17, March 1968. Includes 347 entries.

* R. A. Short and J. Goldberg, A survey of Soviet activities in the design of fault-tolerant digital machines, COMPUTER, vol. 4, 1, Jan-Feb 1971; also appears as Soviet progress in the design of fault-tolerant digital machines, IEEE TRANS. ON COMPUTERS C-20, 11, pp. 1337-1352, November 1971. Includes 714 Soviet entries.

CITED REFERENCES

(Anderson and Metze 73) D. A. Anderson and G. Metze, Design of totally self-checking circuits for m-out-of-n codes, IEEE TRANS. ON COMPUTERS C-22, pp. 293-269, March 1973.

(Avizienis 72) A. Avizienis, The methodology of fault-tolerant computing, USA-JAPAN CONFERENCE, pp. 405-413, 1972.

(Avizienis 71) A. Avizienis, Arithmetic error codes: cost and effectiveness studies for application in digital system design, IEEE TRANS. ON COMPUTERS C-20, pp. 1322-31, November 1971.

(Avizienis et al. 71) A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr and D. K. Rubin, The STAR (Self-Testing and Repairing) computer: an investigation of the theory and practice of fault-tolerant computer design, IEEE TRANS. COMP. C-20, pp. 1312-21, Noember 1971.

(Baer 73) J. L. Baer, A survey of some theoretical aspects of multiprocessing, COMPUTING SURVEYS 5, pp. 31-80, March 1973.

(Berlekamp 68) E. R. Berlekamp, Algebraic Coding Theory, McGraw-Hill, NY, NY, 1968.

(Bossen 70) D. C. Bossen, b-adjacent error-correction, IBM JOURNAL OF RESEARCH AND DEVELOPMENT 14, 4, pp. 402-409, 1970.

(Bow 73) R. T. Bow, Codes for high speed arithmetic and burst correction, Report R-597, Coord. Science Lab., Univ. Illinois, Jan 1973.

(Carter et al. 70a) W. C. Carter, et al., Design Techniques for Modular Architecture for Reliable Computer Systems, IBM Report 70-208-0002 under Contract NAS8-24883, Yorktown Hts. NY, March 26, 1970.

(Carter et al. 70b) W. C. Carter, D. C. Jessep and A. B. Wadia, Error-free decoding for failure-tolerant memories, PROC. IEEE INTL. COMPUTER GROUP CONF., Washington D. C., pp. 229-239, June 1970.

(Carter et al. 71a) W. C. Carter, D. C. Jessep, A. B. Wadia, P. R. Schnieder and W. G. Bouricius, Logic Design for Dynamic and Interactive Recovery, IEEE TRANS. ON COMPUTERS C-20, pp. 1300-05, November 1971.

(Carter et al. 71b) W. C. Carter, K. A. Duke and D. C. Jessep, A simple self-testing decoder checking circuit, IEEE TRANS. ON COMPUTERS C-20, pp. 1413-14, November 1971.

(Carter et al. 72a) W. C. Carter, A. B. Wadia and D. C. Jessep, Jr., Computer error control by testable morphic Boolean funcions - a way of removing hardcore, DIGEST 1972 INT. SYMP. FAULT-TOLERANT COMPUTING, IEEE Computer Soc., Waltham, Mass, pp. 154-159, June 1972.

(Carter et al. 72b) W. C. Carter, K. A. Duke, and D. C. Jessep, Jr., Lookaside techniques for minimum circuit memory translators, IEEE TRANS. ON COMPUTERS C-22, pp. 283-289, March 1973.

(Dijkstra 65) E. W. Dijkstra, Cooperating sequential processes, Report EWD 123, Math. Dept., Techn. Univ. Eindhoven, The Netherlands, Sep 1965. Also in: Programming Languages (ed. F. Genuys), Academic Press, London, 1968.

(Dijkstra 68) E. W. Dijkstra, The structure of the "THE" multiprogramming system, COMM. OF THE ACM 11, pp. 341-346, May 1968.

(Dijkstra 69) E. W. Dijkstra, Notes on structured programming, Report EWD 249, Math. Dept., Techn. Univ. Eindhoven, The Netherlands, August 1969. Also in Structured Programming, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press, London and New York, 1972.

(Elias 58) P. Elias, Computation in the presence of noise, IBM JOURNAL OF RESEARCH AND DEVELOPMENT 2, pp. 346-353, October 1958.

(Elspas 62) B. Elspas, Design and Instrumentation of Error-Correcting Codes, SRI Final Report RADC-TDR-62-511, Contract AF 30(602)-23277, October 1962 (AD-299 957).

(Elspas and Short 62) B. Elspas and R. A. Short, A note on optimum burst-error correcting codes, IRE TRANS. ON INFORMATION THEORY IT-8, pp. 39-42, January 1962.

(Feustel 73) E. A. Feustel, On the advantages of tagged architecture, IEEE TRANS. ON COMPUTERS C-22, pp. 644-656, July 1973.

(Forbes et al 65) R. E. Forbes et al, A self-diagnosable computer, AFIPS PROC. FALL JOINT COMPUTER CONF., pp. 1073-86, 1965.

(Goldberg et al. 73) J. Goldberg, K. N. Levitt and J. H. Wensley, An organization for a highly survivable memory, DIGEST 1973 INT. SYMP. FAULT-TOLERANT COMPUTING, Palo Alto, pp. 59-64, June 20-22, 1973.

(Holt 72) R. C. Holt, Some deadlock properties of computer systems, COMPUTING SURVEYS 4, pp. 179-196, 1972.

(Heart 73) F. E. Heart, A new minicomputer/multiprocessor for the ARPA Network, PROC. OF THE NATIONAL COMPUTER CONF., New York, NY, June 1973.

(Hong and Patel 72) S. J. Hong and A. M. Patel, A general class of maximal codes for computer applications, IEEE TRANS. ON COMPUTERS C-21, pp. 1322-31, December 1972.

(Horning and Randell 73) J. J. Horning and B. Randell, Process structuring, COMPUTING SURVEYS 5, pp. 5-30, March 1973.

(Kautz 62) W. H. Kautz, Codes and codmng circuitry for automatic error correction within digital systems, in Redundancy Techniques for Computing Systems, ed. Wilcox and Mann, Spartan Books, pp. 152-195, 1962; esp. p. 189.

(Kautz and Levitt 72) K. N. Levitt and W. H. Kautz, Cellular arrays for the solution of graph problems, COMM. OF THE ACM 15, pp. 789-801, September 1972.

(Ko 73) D. C.-C. Ko, Self-Checking of Multi-Output Combinational Circuits Using Forced-Parity Techniques, USCEE Report 451, Univ. Southern Cal., Electronic Sciences Lab., June 1973.

(Kuo and Abramson 73) F. F. Kuo and N. Abramson, Computer-Communication Networks, Prentice-Hall, 1973.

(Langdon and Tang 70) G. G. Langdon and C. K. Tang, Concurrent error detection for group look-ahead binary adders, IBM JOURNAL OF RESEARCH AND DEVELOPMENT 14, September 1970.

(Laws 72) B. A. Laws, A ROM decoder for the (15,13) Reed-Solomon Code, Xerox Research Center, Palo Alto, Ca, 1972.

(Levitt et al. 68) K. N. Levitt, M. W. Green and Jack Goldberg, A study of the data commutation problems in a self-repairable multiprocessor, AFIPS PROC. OF THE SPRING JOINT COMPUTER CONF., pp. 515-527, 1968.

(Lofgren 58) L. Lofgren, Automata of high complexity and methods of increasing their reliability by redundancy, INFORMATION AND CONTROL 1, pp. 126-147, 1958.

(Mitarai and McCluskey 72) H. Mitarai and E. J. McCluskey, Design of a parallel encoder/decoder for the Hamming code, using ROM, Tech. Rpt. 36, Dig. Syst. Lab., Stanford Electr. Labs., Stanford Univ., Ca, June 1972.

(Monteiro and Rao 72) P. Monteiro and T. R. N. Rao, A residue checker for arithmetic and logical operations, DIGEST 1972 INT. SYMP. FAULT-TOLERANT COMPUTING, pp. 8-13, June 19-21, 1972.

(Neumann 69) The role of motherhood in the pop art of system programming, ACM SECOND SYMP. ON OPERATING SYSTEMS PRINCIPLES, Princeton NJ, pp. 13-18, October 20-23, 1969.

(Neumann 72) P. G. Neumann, A hierarchical framework for fault-tolerant computing systems, DIGEST IEEE COMPUTER SOCIETY CONFERENCE (COMPCON), San Francisco, Ca., pp. 337-340, September 12-14, 1972.

(Neumann 73) System design for computer networks, Chapter 2 of Computer-Communication Networks, ed. Kuo and Abramson, Prentice-Hall, pp. 29-81, 1973.

(Neumann and Rao 73) P. G. Neumann and T. R. N. Rao, Error correction in byte-organized arithmetic processors, DIGEST 1973 INT. SYMP. FAULT-TOLERANT COMPUTING, pp. 53-58, June 20-22, 1973.

(Ore 63) Graphs and Their Uses, Random House, New York, NY, 1973. See p. 44.

(Parhami and Avizienis 73) B. Parhami and A. Avizienis, Application of arithmetic error codes for checking of mass memories, DIGEST 1973 INT. SYMP. FAULT-TOLERANT COMPUTING, pp. 47-51, June 20-22, 1973.

(Parnas 72) D. L. Parnas, On the criteria to be used in decomposing systems into modules, COMM. OF THE ACM 15, pp. 1053-58, December 1972.

(Peterson and Weldon 72) Error-Correcting Codes, 2nd. ed., MIT Press, Cambridge, Mass, 1972.

(Pierce 65) W. H. Pierce, Failure-Tolerant Computer Design, Academic Press, New York, NY, 1965.

(Rao 70) T. R. N. Rao, Biresidue error correcting codes for computer arithmetic, IEEE TRANS. ON COMPUTERS C-19, pp. 398-402, May 1970.

(Rohr 73) J. A. Rohr, STAREX self-repair routines: software recovery in the JPL-STAR Computer, DIGEST 1973 INT. SYMP. FAULT-TOLERANT COMPUTING, pp. 11-16, Jun 20-22, 1973.

(Schroeder and Saltzer 72) M. D. Schroeder and J. H. Saltzer, A hardware architecture for implementing protection rings, COMM. OF THE ACM 15, pp. 157-170, March 1972.

(Sevcik et al. 72) K. C. Sevcik, J. W. Atwood, M. S. Grushcow, R. C. Holt, J. J. Horning, D. Tsichritzis, Project SUE as a learning experience, AFIPS PROC. OF THE FALL JOINT COMPUTER CONF. 41, pp. 331-338, 1972.

(Siewiorek and Ingle 73) Private communication...draft document.

(Simon 62) H. A. Simon, The architecture of complexity, PROC. AM. PHIL. SOC. 106, pp. 467-482, December 1962. Also in: The Sciences of the Artificial, MIT Press, Cambridge Mass, 1969.

(Spier and Organick 69) The Multics interprocess communication facility, ACM SECOND SYMP. ON OPERATING SYSTEMS PRINCIPLES, Princeton NJ, pp. 83-91, October 20-23, 1969.

(Stern 73) Organization and operation of the Multics backup system, Multics Checkout Bulletin 1077, MIT Project MAC, March 23, 1973.

(Stern and Van Vleck 73) Proposed improvements to the Multics backup system, Multics Checkout Bulletin 1076, MIT Project MAC, March 19, 1973.

(Stiffler 73) J. J. Stiffler, The SERF Fault-Tolerant Computer. Part I: Conceptual design, DIGEST 1973 INT. SYMP. FAULT-TOLERANT COMPUTING, pp. 23-26, June 20-22, 1973.

(Turn 72) R. Turn, Air Force Command and Control Information Processing in the 1980's: Trends in Hardware Technology, RAND Report R-1011-PR, October 1972.

(Wensley 72) J. H. Wensley, SIFT - Software Implemented Fault Tolerance, AFIPS PROC. OF THE FALL JOINT COMPUTER CONF., pp. 243-253, 1972.

(Wensley et al. 73) J. H. Wensley, K. N. Levitt, M. W. Green, P. G. Neumann, J. Goldberg, Fault Tolerant Architectures for an Airborne Digital Computer, Stanford Research Institute, Report of Task I, Contract NAS1-10920, July 24 1972 (Final Report -- preliminary version).

(Yourdon 72) E. Yourdon, Reliability of Real-Time Systems, Prentice-Hall, 1972.

(Zipf 49) G. K. Zipf, Human Behavior and the Principle of Least Effort, Addison-Wesley, Reading, Mass, 1949.

APPENDIX 1
CENSUS OF FAULT-TOLERANT COMPUTING SYSTEMS

This is a brief summary of systems and system designs providing significant fault-tolerance and/or availability. Those systems indicated by "(A2)" are considered in greater detail in the Survey of Fault Tolerant Computing Systems (Appendix 2), where references are included. Terse references are given here. Several systems are described in what is referred to here as the "Intermetrics Report" (J. S. Miller et al., Multiprocessor Computer Study, Final Report, Contract NAS 9-9763, Intermetrics, Inc., Cambridge, Mass, March, 1970). Abbreviations: P=Processor, M=Memory, (S)EC=(single) error correction, (D)ED=(double) error detection. A measure of the hardware overhead for fault tolerance is given as that percent of all hardware dedicated to fault-tolerance (on an approximate cost basis).

A. GENERAL-PURPOSE COMPUTING UTILITIES, generally good availability, human users, modest reliability, maintenance permitted.

1(A2). Multics, MIT (F. J. Corbato) and Honeywell, Cambridge, Mass; ARPA-funded development, now Honeywell product. See E. I. Organick, The Multics System, MIT Press 1972.
* General-purpose computing utility (time-sharing, batch), with high availability and file integrity. Four installations currently exist.
* 1-7 P (Honeywell 6180s), typically 2P, multiprocessed multiprogramming totally reentrant procedure, virtual memory, manual reconfiguration of multiple P and M during operation, extensive isolation via the ring mechanism for protection and via file system access control, incremental file backup, variable-depth system recovery, redundancy in the file directory structure. SED in main memory. Significant security. Hardware negligibly redundant. Software variably redundant, e.g., 20% overhead in time for guaranteed 30-minute lag backup.

2(A2). PRIME, University of California at Berkeley (H. Baskin); ARPA.
* Reliable, secure, modest computer utility, high availability. In development.
* 5 P (design practical for 3 P to 8 P), with highly restricted possible connectivity among M, P and disk, strict isolation with no memory sharing or multiprogramming, "spontaneous" reconfiguration via a reliable self-checking switch. Hardware less than 10% redundant, software less than 10% redundant in time.

3(A2). Carnegie-Mellon University; ARPA.
* Research system development with applications to ARPA speech understanding project; in design. 2x2 version exists.
* 16 P x 16 M (modified PDP 11s), with reliable crossbar switch. Hard and soft reconfigurability, with widely varying operating modes. Hardware less than 5% redundant.

4. University of Newcastle-on-Tyne, Engl.; Scientific Research Council.
* General computing; in design
* PDP 11s

Note. Burroughs B7700 and IBM System/370 have significant hardware facilities for fault tolerance. Also, various commercial time sharing services gain availability (but not necessarily reliability) by having multiple P, M and secondary memory units cross-switchable.

B. GROUND-BASED SPECIAL PURPOSE SYSTEMS, controlling the environment (or vice versa), generally higher reliability and availability, often tighter real-time constraints than those above, usually maintainable.

5(A2). ESS (Electronic Switching Systems), Bell Labs, Naperville, Ill.
* Telephone switching system; long-term continuous system availability, with occasional errors supposedly tolerable to customers. Over 200 Number 1 ESS in operation, many more Number 2 ESS, TSPS.
* 2 P (1 functional, 1 standby checking and diagnosis), automatic reconfiguration. Separate nonalterable program store with SEC. 50% of

all programs are diagnostics. Millions of hours of experience have aided in improving hardware and software reliability. People problems still difficult (operations, maintenance). About 50% redundant in hardware. Storage for software due to fault tolerance also significant -- half of all programs.

6(A2). PLESSEY System 250, The Plessey Co., Ltd., Taplow England.
* Telephone and data switching, long-term continuous availability, modular expandability. Prototype end of 1971.
* 1-16 P, 1-30 M, each 16-64K. Multiprocessing, multiprogramming, virtual memory, totally reentrant, capability-based protection and sharing. Continued operation via reconfigurability with everything multiply available. Extensive hardware fault detection, operating system consistency checks, background test routines. Hierarchical software recovery. Hardware 20-50% redundant, depending on use.

7(A2). High-speed modular interface message processor (IMP) for the ARPANET, Bolt Beranek & Newman, Cambridge, Mass.
* Store and foreward for interhost message switching. High availability. Reliability largely left to hosts.
* 1-14 P initially, each with 4K M. Smoothly degradable, e.g., in 2 P units. Distributed power, cooling.

8(A2). CLC, Bell Labs, Whippany NJ; ABMDA (Safeguard)
* Safeguard missile defense; continuous availability when (and if) required. In development since mid-60s.
* Up to 10 P, multiprocessed, on-line sparing, separate program memory not writeable; program retry; ED via four-bit check on 64-bit words.

9. FAA (Federal Aviation Adm.), IBM. See IBM Sys J., vol 6, no 2, 1967.
* Air traffic control, long-term continuous availability. Untolerated nontransient errors can be disastrous. About 20 systems at ATC centers covering the continental United States.
* Up to 4 P (IBM 9020), up to 12 M. Program-controlled error analysis and reconfiguration, gracefully deconfigurable. 5-second battery backup power supply. Relies heavily on good available field engineers.

10. Flight Plan Processing System, Marconi Radar Systems Ltd., Chelmsford, England.
* Real-time air-traffic control. At most one 30-sec interrupt per year, at most one longer interruption in 5 years, immune to power failures, fast repair of faulty equipment.
* 3 P (MYRIAD)

11. MDS-2 (Market Data System), New York Stock Exchange
* Stock trading ticker control. Near-continuous availability, no transaction losses permitted. Operational August 1972. Precursor MDS-1 operational for 7 years.
* 3 P (360/50), 2 multiprocessing with shared M & LCS (but 1 P basically monitoring), 3rd P normally spare (running background jobs), extensive program checking. Highly replicated peripherals (I/O, disks, etc.)

12(A2). COMEX, Pacific Coast Stock Exchange
* Stock trading message switching; near-continuous availability, no transaction losses permitted, small real-time lag permitted. Operational since 1969.
* 2 complete systems (each has 360/50 plus 2 PDP 8s), one in San Francisco, one in Los Angeles, capable of running separately or cross-switched (interconfigurable).

13. NASDAQ, National Association of Securites Dealers Automated Quotations; See Datamation, March 1972, pp. 42-45.
* On-line interactive system to facilitate trading of OTC securities; high availability; operational since end of 1971.
* 2 P (1108s), multiprocessing under EXEC 8, capable of running simplex. Dual records in file structure, automatic recovery techniques.

14. Standard Telecommunications Lab, Harlow, England. See Electrical Review, 6 Feb 1970, pp. 1-3.
* Real-time control
* 1 P, SEC/DED in M, in transfers, and in I-O; duplication of punch/reader and of M access switches; triplication of control and of function unit. 52% of hardware due to fault tolerance.

15. Foxboro 88, Foxboro Corp. Process control using 2 P (PDP 8)

C. AERO-SPACE SYSTEMS, usually with ultra-high reliability and availability requirements, usually critical real-time contraints, human maintenance usually not possible. At least the first four efforts have resulted in prototype systems. The remaining efforts represent mostly designs in various stages of completion.

16(A2). JPL-STAR, JPL, Pasadena Cal (A. Avizienis); NASA
* Unmanned outer-space travel computer, long-life availability without maintenance. Prototype in operation since 1969.
* 1 P (uniprocessing), heavy use of coding (residue checking for SED in memory and arithmetic, ED in op codes), duplicated logic operations, triplicated monitoring and control (TARP = test and repair processor), replacement by spares via power switching. User-provided rollback points. 60% of hardware due to fault tolerance.

17(A2). MECRA, Electronique Marcel Dassault, St. Cloud, France; DRME
* General-purpose design for special-purpose applications, including aerospace. Prototype now working.
* Duplex arithmetic, Hamming code (7,4) as DED on coded decimal representations (with six unused combinations), sparing, microprogrammable reconfiguration. About 66% redundant.

18(A2). ACGN, CERBERUS, etc., MIT Draper Lab, Cambridge, Mass (A.L. Hopkins, Jr.); NASA/MSC.
* Apollo manned space on-board control, very high reliability during the mission without maintenance. Prototype exists.
* At least 1 processing unit (up to 6), multiprocessing among processing units, replication within each processing unit and within memories (without coding). Two concepts:
(a) duplexed processing units, triplexed scratchpad memories, triplexed memories and buses, with spares;
(b) triplexed processing-scratchpad units.
About 80% redundant.

19(A2). MDC (Modular digital computer), IBM Yorktown Hts, NY,; NASA-Huntsville.

* Modular system, wide range of high-reliability applications; design only

* m P, multiprocessing and replication as well. FO-FO-FS (fail-operational on first and second faults, fail-safe on the third) in 4 P fault-tolerant mode, detection mode also possible. Microdiagnostics, b-adjacent multiple errors handled in M, extensive self-checking.

20(A2). MSC (Modular spacecraft computer), Ultrasystems (Newport Beach Ca) and Raytheon (Waltham Ma); SAMSO/SYT (Los Angeles Ca)

* Reconfigurable guidance and control, space shuttle use; long-life reliability.

* The Raytheon entry in this effort has 1 P, identical subP and subM reliably switchable with sparing. SEC in M plus 3 spare bits reliably switchable via "rippler", burst-error detection in mass M, triplicated control, duplicated configuration control.

* The Ultrasystems entry is similar to the JPL STAR.

21(A2). SIFT, Software implemented fault tolerance, SRI (John Wensley); NASA-Langley

* Airborne control (commercial aviation); availability of correct results during flight; some tasks more critical than others, permitting slight degradation of less critical tasks. Design only (see 1972 FJCC).

* Multiprocessing with variable software replication, dependent on application program (software reconfigurable). Fault tolerance via software can avoid special hardware, permits use of existing designs. Connectivity is restricted: P can modify only its own M, can read others, limits fault propagation. Executive uses the same fault-tolerance procedures as application programs. About 75% redundant.

22(A2). ARMMS, Hughes, Fullerton CA (W. L. Martin); NASA-Marshall (MSFC)

* Spaceborne control; long-life reliability

* m P, dynamically reconfigurable, e.g., as independent-process multiprocessing or as replication with sparing. 20%-80% redundant (variable)

23(A2). Intermetrics multiprocessor, Cambridge Mass (J. C. Miller), outgrowth of EXAM; NASA-ERC (Houston)

* Manned orbiting space station
* m P (1 to 8, nominally 3), each P internally duplicated, coding in M (ED), capability for dynamic duplication of critical data words, buffered instruction retry, save within interrupted instruction.

24(A2). Autonetics (N. Am. Rockwell, Anaheim, L. J. Koczela); NASA-MSC

* Space shuttle; long-life reliability
* 4-level redundancy FO-FO-FS (cf. MDC) 80% redundant; less for lower fault tolerance.

25. SIRU (Strapped-down inertial reference unit), MIT Draper Lab (A. L. Hopkins, Jr.). See Intermetrics Report (reference above).

* Apollo guidance. Simple prototype built.
* 2 P (1 as standby), M duplicated.

26. MULTIPAC, General Telephone and Electr., Waltham, Mass; NASA-Ames. See IEEE Trans. Aerospace and Electronic Sys., Sept. 1971, pp. 974-981.

* Data handling for deep-space probes. Long life, but arbitrary outages can usually be tolerated. Design only.
* Up to 5 P, 15 M (4 K each), gracefully degradable to 1 P, 1 M. Manual reconfiguration of software and hardware via ground-based diagnosis, reprogramming, reassembly and transmittal of a new system into space. Maintainable despite wide range of problems.

27. BUCS (bus checker system), SRI (Karl Levitt), NASA-Langley. See SRI Final Report, NAS1-10920, 1973.

* Aircraft control, as in SIFT
* 5-10 (local) P & M units, each duplicated internally, byte coding in central M, bus checker coordinates restart mechanism, periodic diagnoses of M and of unflexed processor functions. About 33% redundant.

28. TOPS, JPL (Gilley). See IEEE Trans. Astr-Aero, Sept. 1970.

* Thermo-electric outerplanet space travel
* Related to JPL-STAR.

29. MFC, Hamilton-Standard; NASA-ERC. See Intermetrics Report.
* Modular flight computer
* 3 P, 3 M, cross-configurable, TMR or 3 P multiprocessor

30. ALPHA, CDC. See Intermetrics Report.

31. AADC, Honeywell; NASA, AADC Naval Air Systems Command. See Intermetrics Report.

32. IRAD, Litton. See Intermetrics Report.

33. SDC-Burroughs; USAF-Wright-Patterson, Multiprocessor

34. S-3, Univac

35. SUMC, RCA Advanced Technology Lab, Camden NJ; NASA Huntsville.
*Space ultra-reliable modular computer, COSMOS technology.

APPENDIX 2
SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS

This appendix presents replies to a questionaire sent to architects of various fault-tolerant computing systems. It is hoped that the questionaire will itself be useful as a descriptive form and that the replies will aid in understanding and comparing the systems included here. To this end the questionaire has been designed to permit a concise description of each system, its goals, its motivations, its principles, its structure, its techniques, and its achievements to date.

The replies given here are included essentially in their entirety. Several significant efforts are unfortunately not represented here, e.g., IBM's FAA system, the New York Stock Exchange System MDS-2, and a system under development at the University of Newcastle-on-Tyne.

The first issue of this survey was distributed informally to conference participants at the Second International Symposium on Fault-Tolerant Computing, Boston, June 19-21, 1972. It supported the panel discussion "Approaches to the Architecture of Fault-Tolerant Computing", chaired by Jack Goldberg.

The contents of this appendix are as follows.

SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS--QUESTIONNAIRE
SRI Computer Science Group, June 1972

1. IDENTIFICATION of the system
1.1. NAME: What is the relevant name of the system?

1.2. RESPONSIBILITY: What is the responsible organization?

1.3. SUPPORT: What are the sources of support?

1.4. PARTICIPANTS: Who (and what organizations, if relevant) are the principal participants?

1.5. START: What was the date of conception?

1.6. COMPLETION: What was, or is expected to be, the completion date? (Specify prototype acceptance date, or design completion date if design only.)

1.7. BIBLIOGRAPHY: What are the most relevant references?

2. MOTIVATION for the system
2.1. PURPOSE: What is the main purpose of the system (e.g., general-purpose computing, real-time air-traffic control, store-and-forward)?

2.2. PHYSICAL ENVIRONMENT: Where does the system operate (e.g., ground-based, airborne, spaceborne)?

2.3. COMPUTING ENVIRONMENT: How does the system relate computationally to its environment (e.g., locally, remotely, via a network, interactively, via peripherals, with human users)?

2.4. COMPUTING OBJECTIVES: What are the specific computing objectives, regarding capability, capacity, performance (throughput or response), configuration scaleability, maximum real-time delays, etc. (as relevant)?

2.5. RELIABILITY OBJECTIVES: What are the specific system reliability objectives, with respect to desired availability during what period, minimum time to system failure, maximum permitted duration of outage, etc.?

2.6. DYNAMIC VARIABILITY: How may these objectives vary during operation? (E.g., how may performance degrade? May performance be exchanged for increased reliability?)

2.7. PENALTIES: What are the penalties arising from faulty operation? (Possible examples include loss of life, badly decreased performance, the necessity of manual intervention, loss of revenue, etc.)

2.8. CONSTRAINTS: What explicit physical constraints exist (e.g., with respect to size, weight, power, cost)?

2.9. TRADEOFFS: What critical tradeoffs exist among the objectives?

3. DESCRIPTION of the system
3.1. ARCHITECTURE
3.1.1. CONFIGURATIONS
3.1.1.1. INTERCONNECTIVITY: What is the basic configuration, and what restrictions exist on interconnectivity? (You may choose to include a block diagram, a PMS diagram a la Bell and Newell, or other useful representation.)

3.1.1.2. RANGE: What is the range over which configurations are sensible (minimum to maximum), e.g., how many processors, how many memory modules (of what size and word length, and with what restrictions if any), etc.?

3.1.1.3. CAPABILITY: What is the effective computing power of the smallest sensible configuration in 3.1.1.2? Please compare it roughly with a well-known system (e.g., 360/40, 65, 195), and cite a ball-park figure for the number of additions per second. Capability required for fault-tolerance should not be included.

3.1.2. EXECUTIVE and operating system
3.1.2.1. MODES of operation: How does the system operate? (E.g.,is each processor multiprogrammable? Is independent-process multiprocessing possible? Is cooperative-process multiprogrammed multiprocessing possible?)

3.1.2.2. SOFTWARE organization: What is the structure of the system software? How is it distributed with respect to the hardware?

3.2. FAULT TOLERANCE
3.2.1. FAULTS TOLERATED: What faults are tolerated by the system, with what resulting effects on system behavior?

3.2.2. FAULTS NOT TOLERATED: What faults cannot be tolerated by the system, and what are the corresponding effects? Identify the weakest links.

NOTE: Faults may be characterized in many ways, including type (e.g., faulty hardware at various levels such as a chip, module, bus, power supply, arithmetic unit, processor, memory; faulty software such as in the executive, in a compiler, or in an applications program; faulty usage and bad inputs), nature (e.g., timing considerations, old age, various physical phenomena), duration and frequency (e.g., one-shot, recurrent, permanent), scope (e.g., isolated faults, correlated or independent multiple faults, with varying degrees of propagation), effect (random, predictable), etc.

3.2.3. TECHNIQUES: What basic techniques are employed to provide fault-tolerant capability, and when, where, and how are they used? Include hardware and software techniques.

NOTE: Applicable techniques include (possibly in combination) replication (e.g., triple-modular redundancy at various levels, redundant computations using independent algorithms), coding (e.g., error-detecting or -correcting codes on a bus, in memory, in arithmetic), repetition and rollback, reconfiguration (including removal without replacement and replacement with spares), diagnostics (e.g., stand-alone, on-line, interactive; preventive, emergency; remote, local), protection (of processes, data, programs, etc.), and outside intervention (human or otherwise). These techniques may be used statically (e.g., always invoked) or dynamically (e.g., configured as needed); at various module levels in hardware and software; in combination with certain events and with certain other techniques.

3.3. NOVELTY: What are the most unusual design features?

3.4. INFLUENCES: What other efforts (systems, research) have had an influence on your system design?

3.5. HARD-CORE: If there is a concept of "hard-core" in your system, what is its significance? (Please define your concept.)

4. JUSTIFICATION for the system
4.1. RELIABILITY EVALUATION: How is reliability estimated and/or demonstrated (e.g., via analysis, simulation, stimulation of faults, theoretical arguments)?

4.2. COMPLETENESS OF EVALUATION: How complete is your design evaluation?

4.3. OVERHEAD: What percentage(s) of total system resources do you attribute to the achievement of fault-tolerance? (Consider cost (logic, execution time, memory, etc., as applicable.)

4.4. APPLICABILITY: What is the potential scope of applicability beyond that stated in sections 2.1 - 2.4 above?

4.5. EXTENDABILITY: In what ways could the system design be advantageously extended, with what increase in cost, and to what effect?

4.6. CRITICALITIES: How critically do the design choices match the design goals? (E.g., could slight changes in goals result in great savings in design, implementation, and/or operation? Is multiprogramming or multiprocessing critical? Is the choice of hardware critical?)

4.7. IMPLICATIONS: What special requirements (if any) does the basic design impose (e.g., on the hardware designers, on the software developers, on users and maintainers)?

5. CONCLUSIONS
5.1. STATUS: What is the current status of the system?

5.2. EXPERIENCE: What conclusions can you reach based on your experience with the system to date (e.g., in design, implementation and operation)?

5.3. FUTURE: What is planned for future development or use of the system?

5.4. ADVANCES: What developments (theoretical or practical) would be desirable for significantly advancing the state of the art in fault-tolerant computing?

6. COMMENTS (Please include any comments on your system, on this questionnaire, etc. which you would like to add. Opinions, prejudices and philosophies are welcomed.

SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS

L. J. Koczela, North American Rockwell Corp.
3370 Miraloma Avenue, Anaheim, California 92803, May 1972

1. IDENTIFICATION
1.1. NAME: A Three Feilure Tolersnt Computer System

1.2. RESPONSIBILITY: Electronica Group, North American Rockwell Corp.

1.3. SUPPORT: Manned Spacecraft Center, NASA

1.4. PARTICIPANTS: L. J. Koczela, J. Jurison, O. Brosius - North American Rockwell; P. Sollock - NASA.

1.5. START: 1/1/70

1.6. COMPLETION: 1/1/71 (design concept)

1.7. BIBLIOGRAPHY: A Three Failure Tolerant Computer System, IEEE Trans. on Computers, November 1971

2. MOTIVATION
2.1. PURPOSE: Real-Time Central Guidance and Control Computer

2.2. PHYSICAL ENVIRONMENT: Spaceborne

2.3. COMPUTING ENVIRONMENT: The computer aystem interacts with avionics subsystems via a multiplexed data bus.

2.4. COMPUTING OBJECTIVES: 30,000 words of memory; 500,000 operations/second speed

2.5. RELIABILITY OBJECTIVES: Must tolerate first two failures with no degradation in performance and third failure with no degradation in safety.

2.6. OYNAMIC VARIABILITY: Third failure could have less computational capacity.

2.7. PENALTIES: Would require manual intervention with possible loas of life.

2.8. CONSTRAINTS: No physicsl constraints but a relative weighting of importance between physical parameters.

2.9. TRADEOFFS: Size, weight and power least important.

3. OESCRIPTION
3.1. ARCHITECTURE
3.1.1. CONFIGURATIONS
3.1.1.1. INTERCONNECTIVITY: Four redundant computers interconnected by four voter switches at their 1/0 channels.

3.1.1.2. RANGE: 2 - 6 CPUa, no reatrictions on word length.

3.1.1.3. CAPABILITY: 500,000 operations/second

3.1.2. EXECUTIVE
3.1.2.1. MODES: The executive may operate the redundant computers in many modes of operation: non-redundant independent computers, multi-programmed, multi-computer, and various combinations of redundancy such as comparison, voting, etc.

3.1.2.2. SOFTWARE: Software control is equally distributed among the redundant computers - no central control exists.

3.2. FAULT TOLERANCE
3.2.1. FAULTS TOLERATEO: Any 3 faults. A fault can range from a single circuit elemsnt to a complete moduls such as a CPU failing. A failure has no effect on system behavior. The system sctually tolerate more than three faults of many different types but it will tolerate at least sny three fsults.

3.2.2. FAULTS NOT TOLERATED: Softwsre fsults that are not caught in debugging.

3.2.3. TECHNIQUES: The technique used is replication of hardware with quedruple redundancy. Computations are performed redundantly and reconfigurstion is sccomplished without removal ur replacement after failure detection by voting.

3.3. NOVELTY: Through the redundant use of edaptive voters operating on the input/output of redundant computers, any three failure can ba tolersted.

3.4. INFLUENCES: None

3.5. HARD-CORE: No hard core exists.

4. JUSTIFICATION
4.1. RELIABILITY EVALUATION: Extensive fault simulations have been aucceasfully performed.

4.2. COMPLETENESS OF EVALUATION: It is impossible to verify a design goal of 100 percent confidence.

4.3. OVERHEAD: For triple failurs tolerance, about 80%, less for lower failure tolerance.

4.4. APPLICABILITY: To many critical real-time control systems, industrial, space and defense applications.

4.5. EXTENDABILITY: The design can be extended to tolerate different numbers of failures, eg. any two failures, any four failurea, etc.

4.6. CRITICALITIES: Requirement for 100% confidence in tolerating any 3 failures is very critical, lowering to 99 percent or so would reduce complexity and cost.

4.7. IMPLICATIONS: Hardware designers must insure independence of failures at computer I/O interfaces.

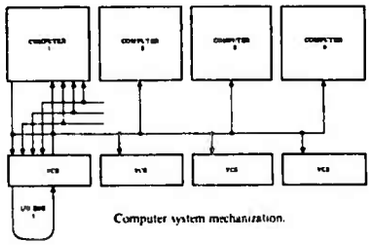
5. CONCLUSIONS
5.1. STATUS: System deeign concept completed, voter-switch deteiled design completed, prototype hardware of voter-switch currently under development.

5.2. EXPERIENCE: A very rigid failure tolerance requirement can be met assuring that a minimum number of failures will be tolerated.

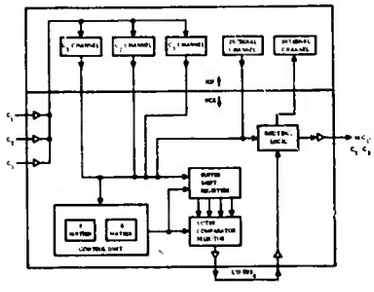
5.3. FUTURE: Possible use on space shuttle program

5.4. ADVANCES: A significant aree that can enhance the atate of the art in designing fault-tolerant computers is analysis of failure modes of components and computer subsystems in depth. Another very important srea is error-free aoftware.

6. COMMENTS: Much of the work on fault-tolerant computers is dedicated to single failures at the gate and circuit level. Unfortunately, in many cases thie is not applicable to real world failures when considering computers mechanized from atate of the art LSI integrated circuits.



Computer system mechanization.



VCS mechanization.

SURVEY OF FAULT TOLERANT COMPUTER SYSTEMS

Algirdas Avizienis
UCLA Computer Science Dept., Los Angeles, CA and
Spacecraft Computer Section, JPL, Pasadena, CA, March 1973

1. IDENTIFICATION
1.1 NAME: JPL-STAR (Self-Testing-And-Repairing) Computer

1.2 RESPONSIBILITY: Spacecraft Computer Section, Astronics Division of the Jet Propulsion Laboratory, Pasadena, California.

1.3 SUPPORT: NASA - Office of Advanced Research and Technology (via JPL)

1.4 PARTICIPANTS: A. Avizienis, D. A. Rennels, J. A. Rohr, F. P. Mathur, G. C. Gilley

1.5 START: 1961

1.6 COMPLETION: Operational - Spring 1969 (laboratory model), modifications continue

1.7 BIBLIOGRAPHY:
*A. Avizienis, et al., The STAR (Self-Testing and Repairing) Computer: An investigation of the theory and practice of fault-tolerant computer design, IEEE Trans. Computer, C-20, pp. 1312-1321, November 1971.

*A. Avizienis, "Design of fault-tolerant computers," FJCC, pp. 733-743, 1967.

*A. Avizienis, "An experimental self-repairing computer," Information Processing, IFIP, Vol. 2, pp. 872-877, 1968.

*A. Avizienis, F. P. Mathur, D. Rennels, and J. A. Rohr, "Automatic maintenance of aerospace computers and spacecraft information and control systems," Proc. AIAA Aerosp. Comput. Syst. Conf., Paper 69-966, pp. 1-11, September 8-10, 1969.

*A. Avizienis, "Concurrent diagnosis of arithmetic processors," Digest of the 1st Annual IEEE Comp. Conf., pp. 34-37, 1967.

*A. Avizienis, "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," IEEE Trans. Comp, C-20,, pp. 1322-1331, Nov 1971.

*F. P. Mathur and A. Avizienis, "Reliability analysis and architecture of a hybrid-redundant digital system: Generalized triple modular redundancy with self-repair," SJCC, pp. 375-383, 1970.

*F. P. Mathur, "On reliability modeling and analysis of ultrareliable fault-tolerant digital systems," IEEE Trans. Comp., C-20, pp. 1376-1382, November 1971.

*G. C. Gilley, "Automatic maintenance of spacecraft systems for long-life, deep-space missions," Ph.D. dissertation, Dept. Comput. Sci., UCLA, September 1970.

*F. P. Mathur, "Reliability estimation procedures and CARE: The computer aided reliability estimation program," Jet Propul. Lab. Quart. Tech. Rev., Vol 1, October 1971.

*A. Avizienis and D. Rennels, "Fault-Tolerance Experiments with the JPL-STAR Computer," Proc. of the Sixth Annual International Conference of the IEEE Computer Society (COMPCON), San Francisco, California, 1972, pp. 321-324.

*A. Avizienis, "Arithmetic Algorithms and Processor Design for Error-Coded Operands," IEEE Transactions on Computers, June 1973.

*G. C. Gilley, "A Fault-Tolerant Spacecraft," Digest of the 1972 International Symposium on Fault-Tolerant Computing, Newton, Mass., June 19-21, 1972, pp. 105-109.

*F. P. Mathur, "Automation of Reliability Evaluation Procedures through CARE--The Computer-Aided Reliability Estimation Program," AFIPS Conference Proceedings (Fall Joint Computer Conference) Vol. 41, Anaheim, California, December 5-7, 1972.

*J. A. Rohr, "System Software for a Fault-Tolerant Digital Computer," Ph.D. Thesis, University of Illinois, Department of Computer Science, Urbana, Illinois, February 1973.

2. MOTIVATION
2.1 PURPOSE: Experimental laboratory GP machine; suitable for spacecraft control

2.2 PHYSICAL ENVIRONMENT: Laboratory environment

2.3 COMPUTING ENVIRONMENT: Local I/O facilities

2.4 COMPETING OBJECTIVES: Capable of automatically maintaining an unmanned spacecraft

2.5 RELIABILITY OBJECTIVES: 100,000 hour survival with 0.95 reliability; tolerance of transient faults; outage for recovery below 50 msec.

2.6 DYNAMIC VARIABILITY: Maximum computing power required at end of mission

2.7 PENALTIES: None for lab model; loss of spacecraft for flight model

2.8 CONSTRAINTS: None for lab model; for the flight model the weight of the subsystem was not to exceed 40 lb. and the power consumption was not to be greater than 40 W.

2.9 TRADEOFFS: None

3. DESCRIPTION
3.1 ARCHITECTURE
3.1.1 CONFIGURATIONS
3.1.1.1 INTERCONNECTIVITY: See Figure

3.1.1.2 RANGE: One processor of each class (operating); 16 memory modules of 4096 words each (maximum operating memory)

3.1.1.3 CAPABILITY: 500 KHz maximum clock rate and byte-serial operation in laboratory model.

3.1.2 EXECUTIVE
3.1.2.1 MODES: The entire set of active STAR computer modules operates as a single, general-purpose computer. The executive implements a two-partition, interrupt-driven, multiprogramming environment on the machine. Four modes of operation under the executive are distinguished. (1) The self-repair mode has highest priority and is entered immediately after hardware self-repair. This mode accomplishes self-repair operations delegated to software such as memory reconfiguration and program resumption. (2) The interrupt mode is used to process interrupts. While in this mode, all lower priority interrupts are inhibited by software. (3) The problem mode is the normal mode of execution for applications programs. All active interrupts are enabled when running in the problem mode. (4) The wait mode is similar to the problem mode except that only low-priority, cyclic programs are run. The registers of wait-mode programs are never saved, and the programs can be resumed at a standard point.

3.1.2.2. SOFTWARE: The software for the STAR computer can be categorized into four efforts: the programming system, the resident executive, the demonstration applications programs, and the spacecraft applications programs. The programming system consists of an assembler, loader, functional simulator, and programming executive. The programming system has been implemented on the UNIVAC 1108. It is used to generate programs for the STAR computer.

The resident executive which has been designed for the STAR computer is called STAREX. The STAREX routines are divided into ten categories: snapshot, self-repair, initialization, scheduling, timing, interrupt handling, library management, facilities management, input-output, and service. The STAREX self-repair routines augment the self-repair hardware facilities by reconfiguring the memory and resuming applications programs after self-repair. STAREX operates in duplicated memory modules and uses a single variable to maintain its rollback point. (The rollback point is the address for program resumption after self-repair.) STAREX also provides facilities for applications programs to establish rollback points.

Demonstration application programs have been developed for demonstrating the STAR computer laboratory breadboard. These programs successfully survive transient and simulated permanent faults and properly resume computation after the fault is removed. These programs establish rollback points by calling the executive routines.

Spacecraft applications programs have been investigated as part of the preliminary studies of the TOPS control computer subsystem which was eventually to be used on board the Grand Tour spacecraft.

3.2 FAULT TOLERANCE

3.2.1 FAULTS TOLERATED: The principal goal of the design is to attain fault tolerance for a variety of faults: transient, permanent, random, and catastrophic.

3.2.2 FAULTS NOT TOLERATED: (a) Transients at a rate higher than allowed by the length of "rollback" segments of programs; (b) shorted bus wires (isolators are employed) or power switch "on" failures.

3.2.3 TECHNIQUES: All machine words (data and instructions) are encoded in error-detecting codes. Fault detection occurs concurrently with program execution. The computer is divided into a set of replacable functional units containing their own instruction decoders and sequence generators. This decentralization allows simple fault-location procedures and simplifies system interfaces.

* Fault detection, recovery, and replacement are carried out by special-purpose hardware. Memory reconfiguration and program resumption are accomplished by the resident executive.

* Transient faults are identified and their effects are corrected by the repetition of a segment of the current program; permanent faults are eliminated by the replacement of faulty functional units.

* The replacement is implemented by power switching: units are removed by turning power off and connected by turning power on. The information lines of all units are permanently connected to the buses through isolating circuits; unpowered units produce only logic "zero" outputs.

* The error-detecting codes are supplemented by monitoring circuits which serve to verify the proper synchronization and internal operation of the functional units.

* The "hard core" test and repair processor (TARP) is protected by triplication and replacement of failed members of the triplet.

3.3 NOVELTY: Power switching, status signals, encoding of instructions emphasis on transient-recovery with program survival.

3.4 INFLUENCES: Theoretical work by Reed and Brimley; Kruus and Seshu; Griesmer, Miller and Roth.

3.5 HARD-CORE: The "hard core" monitor of the STAR system is designated as TARP (test and repair processor) in the Figure. The TARP monitors the operation of the STAR computer by two methods: (1) testing every word sent over the two data buses for validity of its code; and (2) checking the status messages from the functional units for predicted responses.

Three fully powered copies of the TARP are operated at all times together with n standby spares (n = 2 in the present design). The outputs of the TARPs are decided by a 2-out-of-(n+3) threshold vote. When one powered TARP disagrees with the other two, the recovery mode is entered and an attempt is made to set the internal state of the disagreeing unit to match the other two units. If this TARP rollback attempt fails, the disagreeing unit is returned to the standby condition and one of the standby units receives power, goes through the TARP rollback, and joins the powered triplet. A standard rollback then occurs and the resident executive resumes normal program operation. Because of the three unit requirement, design effort has been concentrated on reducing the TARP to the least possible complexity. Experience with the present model has led to several refinements of the design.

The replacement of faulty functional units is commanded by the TARP vote and is implemented by power switching. It offers several advantages over the switching of information lines which connect the units to the bus. The number of switches are reduced to one per unit, power is conserved, and strong isolation is provided for catastrophic failures. Magnetic power switches have been developed which are part of each unit's power supply and are designed to open for most internal failures. The threshold function is inherent in the control windings of the switch. The information lines of each unit are permanently connected to the buses through component-redundant isolation circuits. The signal on a bus is the logic OR of all inputs from the units, and unpowered units produce only logic zero outputs. The power switch and the buses utilize component redundancy for protection against fatal "shorting" failures.

4. JUSTIFICATION

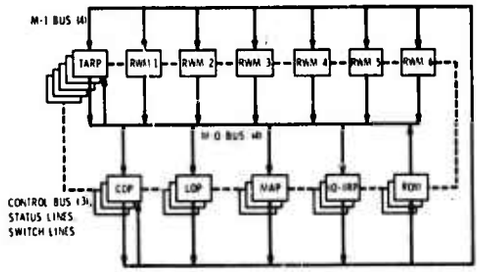
4.1 RELIABILITY EVALUATION: The computing operations for the analysis was done with the aid of the computer-aided reliability estimation (CARE) program, which was developed as a design tool during the reliability study. CARE is a software package developed on the Univac 1108. CARE may be interactively accessed by a designer from a teletype console to calculate his reliability estimates. The input is in the form of a system configuration description followed by queries on the various reliability parameters of interest and their behavior with respect to mission time, fault coverage, failure rates, dormancy factors, allocated spares, and partitioning. The CARE program is extensible, and it may be updated to incorporate new reliability models as they become available. A second set of programs, the Reliability Modeling System (RMS), was developed as a tool in the experimental verification of the STAR breadboard. This set of programs computes the reliability of the various subsystem configurations using "coverage" parameters experimentally obtained by inserting faults into the system. RMS is an interactive system implemented by APL.

4.2 COMPLETENESS OF EVALUATION: Physical fault-injection experiments are currently in progress and are expected to be completed in 1973.

4.3 OVERHEAD: Depends on the number of spares. With one spare for each module, the system is about 60% redundant (i.e., about 150 percent extra cost for fault tolerance).

4.4 APPLICABILITY: Various real-time applications that require very fast recovery.

4.5 EXTENDABILITY: Spare processors could be utilized in a multiprocessor mode. Additional buses and supervisory mechanisms would be required.



STAR computer organization.

COP Control processor, contains the location counter and index registers.
LOP Logic processor, (two copies are powered).
MAP Main arithmetic processor.
ROM READ-ONLY memory, 16,384 permanently stored words.
RWM READ-WRITE memory unit (4096 words, two copies powered, 12 units directly addressable.).
IOP Input/Output processor, contains I/O buffer.
IRP Interrupt processor, handles interrupt request.
TARP Test and repair processor, (three copies powered).

4.6 CRITICALITIES: The design goal was a better understanding of replacement systems. In order to retain contact with the practice of computer design, it was decided to design and construct an experimental general-purpose digital computer which would incorporate dynamic redundancy (i.e., fault detection and replacement of failed subsystems) as an integral part of its structure. The design objectives have been carried out and the system, called the STAR computer, began operation in 1969. The modular nature of the STAR computer has allowed systematic expansion and modifications that are still being continued.

An early objective of the design is to study the class of problems which are encountered in transforming the theoretical model of a self-repairing system into a working computer. State-of-the art integrated circuit and memory technology was employed in the design. This objective appears to have been attained reasonably well.

4.7 IMPLICATIONS: Designers must give (a) advance attention to modularization and coded operands; (b) special software features are needed (see 3.1.2.2); (c) users must observe "rollback" rules in programming.

5. CONCLUSIONS
5.1 STATUS: Operating in laboratory; being extensively tested and modified to improve weaknesses that are uncovered.

5.2 EXPERIENCE: Practical implementation of replacement systems is feasible. Transient faults can be systematically eliminated without program loss. Transient tolerance can be specified in terms of "duration" and "frequency" parameters.

5.3. FUTURE: The research and development program which led to the STAR computer is continuing in several directions. Analysis of automatic maintenance algorithms and design of a command/data bus for their implementation are under intensive study. Other current investigations are concerned with the following areas: (1) hardware-software interaction in a fault-tolerant system with recovery, especially the interaction between the TARP and the resident executive; (2) tuning of the resident executive to optimize performance with regard to rollback, both in the executive and applications programs: (3) studies of advanced recovery techniques, i.e., post- catastrophic restart, TARP replacement schemes, recovery from massive interference, partial utilization of failed units; (4) advanced component technology, especially methods to attain bus and power switch (i.e., hard core) immunity to faults; (5) heuristic studies of fault tolerance by interpretation of extensive experiments with the STAR breadboard as the instrument; (6) design of a second-generation STAR-type computer with universal processor and storage modules, and their implementation by large-scale integration; (7) computational utilization of the spare units for supplemental tasks in a multiprocessing mode.

6. COMMENTS: Design, construction, and testing of laboratory models is critically important to advance the state of the art and to gain acceptance among practitioners of design in industry.

The STAR computer breadboard consists of three Read-Write memory units, one Read-Only memory unit, one copy of each of the processing modules, and one TARP (Test and Repair Processor). The breadboard provides adequate facilities for
* experimental varification of the fault detection, diagnosis, and recovery algorithms employed in this construction, and for
* the development of fault-tolerant software techniques.
The development of the breadboard resulted in a direct confrontation with the technological problem areas in fault-tolerant computing, i.e. busing, isolation, power switching, etc. This resulted in a better understanding of these problems and a set of innovative solutions.

SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS

Jacques J. Delamare, Electronique Marcel Dassault (E.M.D.), 55, quai Carnot, 92 - Saint-Cloud France, June 1972

1. IDENTIFICATION
1.1. NAME: MECRA (Maquette Experimentale de Calculateur a Reconfiguration Automatique).

1.2. RESPONSIBILITY: E.M.D. (Electronique Marcel Dassault).

1.3. SUPPORT: Support has three sources: D.G.R.S.T. (Delegation Generale a la Recherche Scientifique) with preliminary studies; D.R.M.E. (Direction des Recherches et des Moyens d'Essais) with realization of MECRA project; E.M.D. (Electronique Marcel Dassault) in each case.

1.4. PARTICIPANTS: Jacques J. Delamare, Gerard Germain, Jean-Claude R. Charpentier, all of E.M.D., and four researchers from "Centre de Calcul Numerique de Toulouse".

1.5. START: May 1970

1.6. COMPLETION: July 1972, this consists of a demonstration of fault tolerance and reconfiguration capabilities. Evaluation of reliability performance is expected to be in Autumn 1972.

1.7. BIBLIOGRAPHY: "The MECRA: a Self Reconfigurable Computer for Highly Reliable Process", IEEE vol C-20 no. 11, pp. 1382-1388, Nov. 1971. A report also due end of 1972.

2. MOTIVATION
2.1. PURPOSE: The system was conceived for research in fault-tolerant computer architecture, feasibility, and reliability evaluation. The idea for further development is a real-time medium-sized computer for aircraft.

2.2. PHYSICAL ENVIRONMENT: System operates in END laboratories.

2.3. COMPUTING ENVIRONMENT: A single peripheral allows communication with MECRA.

2.4. COMPUTING OBJECTIVES: Main objectives of the project were not computing objectives. However addition and multiplication are performed with 11 decimal digits plus sign operands. Complete addition needs less than 300 microsec. Such delays relate to the cycle time of microprogram memory (1 microsec), to response time of discrete circuits, to unused time intervals in each microinstruction cycle, (allowing hardware modifications), and lastly by the microsoftware package (allowing reconfiguration).

2.5. RELIABILITY OBJECTIVES: Practical experiance and a concrete basis for evaluation such as:
reliability gain with different kinds of redundancy,
hardcore contribution in failure probabilities,
hardcore contribution with different architectures,
reliability gain with reconfiguration,
cost increase in control with reconfigurability,
lost time due to reconfiguration (during and after),
hardcore response time with respect to computing time.
These reliability objectives were only of interest for high probabilities of success (probabilities higher than .9).

2.6. DYNAMIC VARIABILITY: Computing speed but not accuracy may degrade with reconfiguration (20% maximum). Performance cannot be exchanged for increased reliability such as : two processors each one having its own job, switched to parallel processing on the same job and checking one another.

2.7. PENALTIES: Penalties from faulty operation can be of several kinds: /Loss of time due to recovery processes, lessened performance after self-reconfiguration, loss of service./ Manual interventions have not been investigated, but will be necessarily improved as a consequence of self-testing and self-healing capabilities of MECRA.

/SRI note: The text enclosed in slashes is an SRI paraphrase of the original survey response./

2.8. CONSTRAINTS: Circuitry size might not exceed four times the size of the equivalent irredundant computer.

3. DESCRIPTION
3.1 ARCHITECTURE
3.1.1. CONFIGURATIONS
3.1.1.1. INTERCONNECTIVITY: See IEEE paper. The basic configuration is a microprogrammed monoprocessor with a bus architecture. A restriction can be seen here since addresses are binary coded, whereas data are Decimal Hamming coded. This has no importance for the purpose of the project, but would not have been used on a prototype.

3.1.1.2. RANGE: Control Unit Configuration:

Maximum	Minimum
4 counters	3 counters
3 spare counters	0 spare counters
8 registers	6 registers
4 spare registers	0 spare registers
3 multiplication proceasors	1 multiplication processor
2 addition processors	1 addition processor
4 'and ' logic processors	3 or 2
4 'or' logic processors	3 or 2
4 'exclusive or' processors	3 or 2
4 'inverter' blocks	3 or 2

Note: Any logic function can fail completely and can be reconfigured with three other functions. In several cases a failed logic function can be reconfigured with only two other function.
Memory configuration: Three memory blocks - 4 K 16-bit words. Each memory block has its own address decoder circuits. At each memory cycle a 48-bit word is read or written; this word contains two identical words of 24 bits each, so that any one of the three blocks can be declared void and the computer still runs if the other two operate properly. Efficiency of address error detection reaches 50% on each memory block. After any read restore cycle, each eight-bit byte (6 bytes) is checked and is switched or not on busses. Then error detection efficiency is 50% with instructions or microinstruction (if there is only one erroneous bit) and 100% with data (if there is one or two erroneous bit).

3.1.2. EXECUTIVE
3.1.2.1. MODES: MECRA is a monoprocessor.

3.1.2.2. SOFTWARE: There are three working modes on the computer: user mode, test-diagnosis mode, decision and reorganization mode.
a) In the USER mode the computer executes the user program.
b) The TEST-DIAGNOSIS mode is set in motion in two different ways to which two different programs correspond. The first is set in motion by interrupts when a failure has been detected by hardware checkers. The goal of this program is to localize precisely where the failure occured. The second program is set in motion periodically and its purpose is to test the computer with the data configurations which reveal failures best. This program allows detection of the errors which cannot be detected by the hardware checkers (i.e. an erroneous data with correct encoding). These two programs update a status table which contains the status of computer components (failed or not, number of transient failures). They also decide to stop the computer when certain catastrophic failures occur or to set in motion the decision and reorganization mode.
c) In the DECISION AND REORGANIZATION mode, a program analyzes the status word (in the status table) of the component in which one of the two test-diagnosis programs has detected a permanent failure and it decides either to reconfigure or to stop the computer.

3.2. FAULT TOLERANCE
3.2.1. FAULTS TOLERATED: Any single fault is tolerated in memories, arithmetic and logic units (since they are mounted in a duplex scheme) or in logic units (quadded redundancy). Any error detected on the busses, switches the MECRA to interrupt programs, while all writing in memories, registers or counters is inhibited. Multiple errors can also be tolerated in number of cases. Multiple errors can lead to repair or to loss of service as said above (2.7.).

3.2.2. FAULTS NOT TOLERATED: Faults not tolerated include errors in the main control circuit, which leads to a design with an increased degree of microprogramming and minimised control circuits. Also not tolerated are errors undetected at the memory output. Power supply failures have not been investigated in MECRA.

3.2.3. TECHNIQUES: One of the goals of MECRA is an investigation of as many fault-tolerance techniques as possible, such as triple modular redundancy, quadded redundancy, duplex redundancy at very low level (clock) and higher level (memories and arithmetic circuits), random redundancy (counters, registers), error detecting codes (Hamming d = 3) and parity bit, repetition, rollback, reconfiguration with removal without replacement, reconfiguration with replacement, diagnosis - stand-alone, preventive and emergency, local protections of process and data. These techniques are used statically. It does not seem possible to describe these techniques in detail in this paper, since it would require a description of the whole computer. Other techniques were also investigated but not used on MECRA, such as stopping the computer during noisy periods, and control of correct microprogram linking.

3.3. NOVELTY: When the project started, two ideas unusual in the literature were employed in MECRA: address decoder redundancy in memories so as to separate address errors and data errors, single-error-free hard-core.

3.4. INFLUENCES: A synthesis of efforts which came almost exclusively from the U.S.A. - universities, laboratories, and research institutes.

3.5. HARD-CORE: This is defined as a circuit, interconnecting several redundant functions, whatever its own redundancy level (it is a relative concept).

4. JUSTIFICATIONS
4.1. RELIABILITY EVALUATION: Reliability is not demonstrated, it is computed, in two steps using a model. The first step concerns analysis and drawing a network model, the second step concerns random failure assignment into the model. After a great number of trials, the program furnishes results (e.g. curves, marginal probabilities...).

4.2. COMPLETENESS OF EVALUATION: Program evaluation is now being tested.

4.3. OVERHEAD: Approximately 60% to 70% of total system resources are devoted to fault tolerance (same percentage for logic, cost, and time).

4.6. CRITICALITIES: Use of decimal coded characters seems not well-fitted to fault-tolerant computers. This change could result in great savings in design. Other points are not critical.

4.7. IMPLICATIONS: The basic design assumes low-level integrated circuits, with a very small number of different circuits.

5. CONCLUSIONS
5.1. STATUS: The system is now operating and will be delivered in July 72, evaluation will follow during October and November.

5.2. EXPERIENCE: Everything is possible, except, perhaps a sufficiently low cost, and reliable packaging and wiring of components. Note that LSI would put problems to fault-tolerant computers because they need more pins to check redundant functions before connecting all together. This would probably lead to simultaneous use of LSI, MSI and small scale integrated circuits. Component manufacturers have not yet taken into account fault-tolerance constraints, but they will probably do so soon.

5.3. FUTURE: First prototype is projected 1976 - 1977, Current computer is projected 1980, Use: Missiles, aircraft, real-time monoprocessors.

5.4. ADVANCES: Different fault tolerant computers can be roughly compared in terms of reliability versus mission time; but this will fall back to evaluations of components and wiring MTBF. Such data, estimated by constructors, do not seem to give a sufficient common basis for evaluations. Theoretical and conventional data on component MTBF seem to be needed for accurate comparisons among different fault-tolerant computers.

SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS
Barry R. Borgerson, Computer Systems Research Project
University of California, Berkeley, May 1972.

1. IDENTIFICATION
1.1. NAME: PRIME

1.2. RESPONSIBILITY: Computer Systems Research Project (CSRP), U. C. Berkeley

1.3. SUPPORT: ARPA - Contract No. DAHC7D 15 C D724

1.4. PARTICIPANTS: Herbert B. Baskin, Principal Investigator; Roger Roberts, Principal Programmer; Barry R. Borgerson, Head, Hardware R & D.

1.5. START: 7/1/7D

1.6. COMPLETION: First prototype to be running about 9/73

1.7. BIBLIOGRAPHY:

*H. B. Baskin, B. R. Borgerson and R. Roberts, "PRIME - An Architecture for Terminal Oriented Systems," Proceedings of the 1972 SJCC, AFIPS Press pp. 431-437.

*B. R. Borgerson, "A Fail-Softly System for Time Sharing Use," Digest of the 1972 International Fault Tolerant Computing Symposium.

*J. T. Quatse, P. Gaulene and D. Dodge, "The External Access Network of a Modular Computer System," Proceedings of the 1972 SJCC, AFIPS Press, pp. 783-790.

*R. S. Fabry, "Dynamic Verification of Operating System Decisions," CSRP Document No. P-14.1, Univ. California, Berkeley, 11/72 (To be publ. CACM).

*B. R. Borgerson, "Spontaneous Reconfiguration in a Fail-Softly Computer Utility," Digest of DATAFAIR 73, Nottingham England, April 1973.

B. R. *Borgerson, Barry R., "Dynamic Confirmation of System Integrity," FJCC 1972, pp. 89-96.

2. MOTIVATION
2.1. PURPOSE: General-purpose, interactive, multi-access computing.

2.2. PHYSICAL ENVIRONMENT: Ground based

2.3. COMPUTING ENVIRONMENT: Remote access over telephone lines and eventually over the Arpanet.

2.4. COMPUTING OBJECTIVES: This is not the primary motivating area in our system design. We anticipate that the original configuration of PRIME will support about 100 users with a worst case response time of less than two seconds for trivial jobs.

2.5. RELIABILITY OBJECTIVES: Because we will be able to repair units as they become faulty, we are aiming for continuous availability. The system performance should never degrade below 75% of its peak capacity.

2.6. DYNAMIC VARIABLITY: Performance cannot be dynamically traded for reliability. However, provisions may someday be added which will allow dynamically trading performance for intraprocess integrity (See Section 6).

2.7. PENALTIES: The effects of intraprocess data contamination (See Section 3.3.2) due to system failures will strongly depend on the nature and purpose of the process. There seems to be no way to generalize about this. If the system itself were to crash, this would no doubt lead to a loss of revenue if PRIME were transferred to a commercial environment.

2.8. CONSTRAINTS: There are no specific constraints of size, weight, and power. The self-imposed constraint on cost is to try to build a fault-tolerant system that is as close in cost as possible to any current system with comparable power and capabilities.

2.9. TRADEOFFS: (Too complicated to deal with briefly; see Sections 4.4, 4.6 and 6.)

3. DESCRIPTION
3.1. ARCHITECTURE
3.1.1. CONFIGURATION
3.1.1.1. INTERCONNECTIVITY: Figure 1 is a block diagram of PRIME. The Interconnection Network (IN) allows any processor to connect to any disk drive, external device, or other processor. Each processor has three such independent paths into the IN. The IN connectivity remains universal over the different system sizes. Universal switching between all processors and all memory blocks is not provided. Instead, each processor always connects to exactly 64K of memory regardless of the size of the system.

3.1.1.2. RANGE: The PRIME architecture will usefully accommodate from 3 to about 30 processors. Each processor could connect to from 16K to 128K of primary memory. Depending on the type of disk drives used, from 1 to 5 drives per processor would be reasonable. The current system has been designed to operate with from three to eight processors without requiring any additional hardware or software design. Useful memory sizes range from 64K to about 256K. Disk drives range from about six to 24. Each processor to be used in the initial implementation of PRIME will be a Nets 4 (Digital Scientific Corp.). The Nets 4 is a general-purpose, 16-bit, 32-register, 90ns-cycle time microprocessor. The memory is 33 bits wide, about 600 ns cycle, and made from 1024-bit MOS chips. The disk drives are double (track) density 2314-type drives that have been modified to transfer information on two heads at a time. The initial configuration will have five processors, 104K of memory, and 15 disk drives.

3.1.1.3. CAPABILITY: The capability is not accurately known at this time.

3.1.2. EXECUTIVE
3.1.2.1. MODES: At any given time, one processor is designated the Control Processor (CP) while the rest function as Problem Processors (PPs). User processes are run on the PPs. Multiprogramming is not used, but processes are overlap-swapped. In order to achieve a very high interprocess integrity, it was decided never to let two processes share memory; hence, cooperative-process multiprocessing is not possible with PRIME.

3.1.2.2. SOFTWARE: The system software is divided into three sections. There is the Central Control Monitor (CCM) which runs on the Target Machine of the CP; the Extension of the Control Monitor (ECM) which resides directly in the microcode of each processor; and the local Monitor (LM) which runs on the Target Machine in the PPs. The CCM is responsible for scheduling processes, allocating resource, and consummating interprocess message transfers. The ECM includes the disk, terminal, and communication controllers, logic for double-checking critical CCM decisions, bootstrap logic, and some intelligence to deal with reconfiguration. The LM contains the file and working-set management systems. The CCM does not get involved with a process after it has started the process up. The procedure followed by the CCM is to allocate the necessary resources, initiate the roll in, and let the LM and ECM take over from there. The CCM will not get involved again until the process either times out or blocks itself. The LM deals only with user processes; it is completely isolated from the rest of the system. Because of this, users will be free to provide their own LM if they do not like the standard one provided.

3.2. FAULT TOLERANCE
3.2.1. FAULTS TOLERATED: PRIME will tolerate all internal faults. That is, the system is expected to continue operating even in the presence of any arbitrary software or hardware faults. The system will reconfigure to run without any piece of hardware that becomes faulty, and mechanisms exist for limiting the effects of any software fault. PRIME has been designed to provide continuous service to (almost) all terminals. In most cases, a faulty unit will be repaired and returned to service before another failure occurs. However, the system will still continue to operate with a substantial part of the resources removed from active use. The system should almost never degrade to below 75 percent of its maximum capacity. In addition to continuity of some minimum service, interprocess integrity violations are prevented at all times; this includes the relatively unstable periods between the onset of a fault and the detection and isolation of the faulty unit.

3.2.2. FAULTS NOT TOLERATED: Only environmental faults are not tolerated by PRIME. The most common of these faults would be in the A.C. power and air conditioning. Since it is easy to see how to back these resources up, no effort has been made to incorporate fault tolerance with respect to these units within PRIME. While PRIME as a system will continue to run in spite of internal failures, individual processes may occasionally get clobbered. That is, no special provisions have been made in PRIME to guarantee intraprocess integrity. Hence transient failures will frequently cause contamination of information for some process. Also hard failures will often clobber one process before being detected. The most serious disruption will probably occur when a disk drive fails. When this happens, all of the processors that were using that drive will be suspended until an operator can recover their data, either by moving the disk pack to another drive, or recovering from tapes in the unlikely event of a head crash. But even in this worst-case catastrophy, only a small part of the users (about 7 percent in the initial system) will be affected.

3.2.3. TECHNIQUES; The basic system-wide technique used to achieve fault tolerance is to allow the system to degrade gracefully by reconfiguring to run without any faulty units. At the heart of the scheme is a distributed architecture with a multiplicity of all functional units except the IN, which is designed to fail softly on its own. Fault detection is accomplished by a variety of methods that include parity on memory and buses, surveillance tests on each processor after each job step, a double check on all critical system-wide decisions made by the CP, and fault injection in such areas as error detectors and the seldom used reconfiguration logic. After a fault is detected, an initial reconfiguration causes a processor not involved in the detection to become the new CP. This virtual "hard-core" then initiates diagnostics to locate the faulty unit, isolate it, and reconfigure the system to run as efficiently as possible without it. A small amount of dedicated hardware associated with each processor guarantees that the initial reconfiguration will be accomplished properly. It is possible to logically isolate each major unit at its system boundaries so that the system can run fine-mesh diagnostics or exercise the hardware to aid in locating the faulty component. In the case of a failure of the isolation logic, any unit can be dynamically powered down to provide guaranteed isolation from the rest of the system.

3.3. NOVELTY: The distributed nature of the system, including the distributed intelligence in the form of the ECMs, provides a very powerful structure whereby fault tolerance is achieved without the use of any "reliable" hardware. Very high-performance low-cost disk drives have been incorporated in such a way as to allow these devices to be used as second level storage, third level storage, and the swapping medium. By distributing these three functions over many identical physical units, very high availability is achieved at what is actually a lower cost and with higher overall performance than would be possible with three distinct types of units. PRIME automatically responds to faults by reconfiguring to run without the faulty unit. Since there is a multiplicity of all functional units except the IN, it is quite easy to run without any particular unit. Rather than make the IN "reliable," a more economical approach was taken whereby carefully controlled failure modes were designed into it. This results in a failure within the IN manifesting itself as a failure of a small number of ports, which is equivalent to losing whatever is attached to those ports, and the system was already designed to handle that eventuality. The reconfiguration structure is also very interesting. Whenever a failure is detected, an initial reconfiguration takes place which establishes a new processor as the CP. The new CP, which is one not involved in the detection of the fault, is then used as the temporary "hard-core" to initiate diagnostics, locate the fault if indeed one exists, and remove the faulty unit from the system. The distributed intelligence of PRIME has been used to provide double checking on all critical system functions, which in turn guarantees that there will be no interprocess interference. Probably the most unusual general feature of PRIME with respect to fault tolerance is that it is self-diagnosing and self-repairing without incorporating any "hard-core."

3.4. INFLUENCES: Many previous efforts have, of course, influenced us, but no single system stands out as having special influence.

3.5. HARD-CORE: No, there is no "hard-core" in PRIME. Instead, the concept of a "floating hard-core" exists whereby a working processor is passed into service as the Control Processor whenever a malfunction is detected. This is consistent with the overall system philosophy of not having any "reliable" hardware anywhere in the system.

4. JUSTIFICATION

4.1. RELIABILITY EVALUATION: Reliability will be demonstrated by stimulation of faults.

4.3. OVERHEAD: The cost of the additional hardware that has been incorporated in PRIME specifically for fault tolerance is less than 10 percent of the total hardware cost of the system. Less than 10% of each processor's useful time is devoted to fault-tolerant functions, since the surveillance programs are run during what would otherwise be idle time while processes are being swapped.

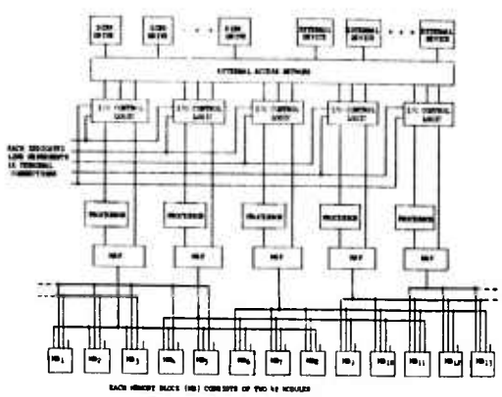
4.4. APPLICABILITY: PRIME has been very carefully designed to perform economically in a particular environment. If it was to be used in another environment, a detailed analysis would have to be performed to determine what changes would have to be made to allow it to perform adequately in the new environment. In particular, many other potential environments would require that steps be taken to guarantee intraprocess integrity.

4.6. CRITICALITIES: The choice of disk drives is quite critical since a low cost/bit is necessary as well as a high bandwidth due to the different functions these drives perform. Since 3330-type drives were not available when this design started, 2314-type drives were selected and modified to transfer at 5MHz. Also, the IN had to be carefully designed with well-specified failure modes. However, the primary memory and the processors are simply "off the shelf" items. As for goals, the decision to not provide intraprocess integrity checks has been carefully exploited in the design of PRIME and has provided a very substantial cost savings.

4.7 IMPLICATIONS: Heavy reliance is placed on periodic checking of hardware rather than concurrent checking. Thus, the ability to inject faults into the appropriate areas has been a difficult requirement placed on all of the hardware designers. The most notable software requirement imposed by the basic design is the clear division of the operating system into three parts, one of which can be furnished by a user. The only significant requirement placed on a user is that he must be aware that no intraprocess integrity checks are made (just like in all current time-sharing systems).

Fig. 1

Block Diagram of the PRIME System



5. CONCLUSIONS

5.1. STATUS: The design of PRIME is about 95 percent completed, and implementation has begun on both the hardware and software. The first version capable of reconfiguring in the presence of a failure should be running by September, 1973.

5.2. EXPERIENCE: The main conclusion that the responden can make regarding the design of PRIME is that by somewhat limiting the goal of the PRIME system, it was possible to create a system that should exhibit excellent fault-tolerant characteristics at a much lower incremental cost than that of any other fault-tolerant system known to him.

5.3. FUTURE: The near future will be devoted to building PRIME. After that, evaluation and tuning will take place with connection to the Arpanet very likely.

5.4. ADVANCES: It seems that the most significant development that would aid the PRIME system would be the availability of a general-purpose, self-checking processor. Since 100 percent self-checkability is extremely difficult to design into a processor, the best course of action here seems to be to wait for LSI processors of sufficient power to be built. These processors should be so inexpensive, compared to the rest of the hardware cost, that running two of them simultaneously and comparing outputs should be a very attractive procedure economically. In fact, the current processing element in PRIME could be broken into several subprocessors: one for communications, one for the disk controller, one for the terminal controller, two for the Target Machine, etc. Probably only the Target Machine processor would have to be duplexed because the others can have independent checks on the validity of their results. With this procedure, intraprocess integrity would be possible at an insignificant incremental cost. For the current version of PRIME, the availability of general procedures for automatically generating test programs would be extremely valuable.

6. COMMENTS: I have experienced a great deal of difficulty locating any other efforts at designing and building what I consider to be truly gracefully degrading self-repairing systems. Most of the effort in fault-tolerant computing to date seems to be centered around military systems, or even moreso, around space exploration systems. This typically dictates that a fixed amount of computing power be made available at all times; hence, the lack of action around fail-softly systems. Of course, by providing fault tolerance through graceful degradation, very substantial cost savings can be realized over the "redundant" methods. In addition to allowing the system's performance to degrade in the presence of faults, we have chosen not to guarantee intraprocess integrity. Also, PRIME uses no "hard-core" to initiate diagnosis or reconfiguration. The combination of these three techniques has allowed us to design a very economical fault-tolerant time-sharing system. There is little doubt that the anticipated degradations will be quite acceptable for a wide range of applications. The lack of intraprocess-integrity guarantees, however, will be a limiting factor in expanding this architecture into other areas. Of course, hardware provisions could be added to guarantee intraprocess integrity, and the resultant system would still be more economical than most other fault-tolerant systems. A more promising approach, and one which we will undoubtedly explore in the reasonably near future, is to leave the hardware as is and run critical programs twice on two different processors. This will allow the system cost to remain very low, and will also allow intraprocess integrity guarantees. Thus, only those processes that that need this guarantee will have to pay for this added feature. A final aspect of the PRIME architecture that should be investigated is whether it can more economically provide a guaranteed computing power in some environments than can be provided by a "redundant" system. It can be overbuilt by an amount sufficient to guarantee that its degraded condition is powerful enough to handle the necessary computing, with background power available most of the time.

SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS

Capt Larry A. Fry, Space and Missile Systems Organization (SAMSO) Los Angeles AFS, CA, February 1973.

1. IDENTIFICATION

1.1. NAME: Modular Spacecraft Computer
1.2. RESPONSIBILITY: SAMSO/DYT, Los Angeles AFS, CA.
1.3. SUPPORT: Not available
1.4. PARTICIPANTS: Raytheon, Sudbury, MA; Ultrasystems, Inc., Newport Beach, CA; Logicon, San Pedro, CA.
1.5. START: Project started mid-1971
1.6. COMPLETION: Logicon is currently implementing interpretative computer simulations of the two MSC designs on the CDC 7600. The architectures and repertoires are being evaluated, along with an intensive study of the fault-tolerance features. Delivery of the ICSs and a study report are due in July.
1.7. BIBLIOGRAPHY: H. Hecht and L. A. Fry, "Fault-Tolerance in the Modular Spacecraft Computer," presented at the 6th International Hawaii Conference, 9-11 January 1973.

2. MOTIVATION

2.1. PURPOSE: Support of all satellite data processing requirements
2.2. PHYSICAL ENVIRONMENT: In satellite
2.3. COMPUTING ENVIRONMENT: Hardwired to environment
2.4. COMPUTING OBJECTIVES: About 200K operations per sec.
2.5. RELIABILITY OBJECTIVES: Nominal probability of survival at the end of five-year life of 0.95; variability achieved by adjusting the number of spares carried.
2.6. DYNAMIC VARIABILITY: Essentially no variability
2.7. PENALTIES: Loss of major satellite functions
2.8. CONSTRAINTS: 25 pounds and 30 watts

3. DESCRIPTION

3.1. ARCHITECTURE
3.1.1. CONFIGURATIONS
3.1.1.1. INTERCONNECTIVITY: Both designs are bus-oriented. Raytheon uses eight general registers; Ultrasystems uses a conventional AC/MQ design.
3.1.1.2. RANGE: Simple processors. Memory is modular in 4K increments, up to 65K 32-bit words. I/O is variable, to suit specific real-time applications.
3.1.1.3. CAPABILITY: Roughly comparable to a 360/40. 500K fixed-point ADDs/sec; 200K floating-point ADDs/sec.
3.1.2. EXECUTIVE
3.1.2.1. MODES: Interruptible but not a true multiprocessor
3.1.2.2. SOFTWARE: Not yet developed. Will have a real-time operating system, including fault-recovery routines.
3.2. FAULT-TOLERANCE
3.2.1. FAULTS TOLERATED: Transient and permanent--all logic types. Also can tolerate some catastrophic faults.
3.2.2. FAULTS NOT TOLERATED: Faults resulting from major physical damage.
3.2.3. TECHNIQUES: Replication; coding; repetition and rollback; and configuration. Techniques used statically and dynamically.
3.3. NOVELTY: Extensive dynamic redundancy
3.4. INFLUENCES: Not available
3.5. HARD-CORE: Configuration Control Unit is triply-modular-redundant, controlling all retries and most reconfigurations.

4. JUSTIFICATION: The failure probability of non-fault-tolerant computers is too high to permit their use as central data processors on long-life spacecraft. Consider a computer using the equivalent of 2500 electronic parts, each of very high reliability such that the part failure rate is 10E-8 per hour. Then the computer failure rate is 25 x 10E-6 per hour. For an exponential failure distribution, the five-year reliability is 0.37, the reciprocal of e; e to the power -(40,000 x 25 x 10E-6).

5. CONCLUSIONS

5.1. STATUS: Peforming interpretive simulation
5.2. EXPERIENCE: Architecture very suitable for intended application.
5.3. FUTURE: Not available
5.4. ADVANCES: Practical design under weight and power constraints.

6. COMMENTS: Raytheon began its design by using duplication as a main approach, while Ultrasystems used arithmetic coding. Neither approach was entirely satisfactory, and it was very enlightening to observe the two designs converge in the course of several iterations. Currently, both employ a balanced mixture of duplication, coding, and TMR. These designs studies also demonstrated the impracticality of the Reed and Brimley approach. Both contractors initially broke up the computer into small modules but found that the switching overhead and attendant complications overshadowed the theoretical reliability improvement. Larger modules are now used, with the computer-on-a-chip in view. An exception was the memory module, where a few spare bit lines seem useful.

SURVEY OF FAULT -TOLERANT COMPUTING

Severo M. Ornstein, Bolt Beranek & Newman, Inc.
Cambridge, Mass 02138, May 1973

1. IDENTIFICATION
1.1. NAME: High Speed Modular IMP (for the ARPANET)
1.2. RESPONSIBILITY: Bolt Baranek & Newman
1.3. SUPPORT: ARPA
1.4. PARTICIPANTS: Frank Heart, Severo Ornstein, William Crowther, Benjamin Barker, Anthony Michal, Michael Kralay, Martin Thrope, all from BBN.
1.5. START: July 1971
1.6. COMPLETION: Prototype summer 1973
1.7. BIBLIOGRAPHY: F. E. Heart, A New Minicomputer/ Multiprocessor for the ARPA Network, Proceedings of the National Computer Conference, New York, N.Y., June 1973.

2. MOTIVATION
2.1. PURPOSE: Store & Forward Message Processor--High Speed IMP Modular Version
2.2. ENVIRONMENT: Ground-based. Remote diagnosis, restart.
2.3. COMPUTING OBJECTIVES: A variable sized nodal element in a nationwide (and some overseas) computer network.
2.4. COMPUTING OBJECTIVES: Throughput capability of about 10 megabits of traffic. Computing power to be 10 times that of a standard mini (such as the Honeywell 516).
2.5. RELIABILITY OBJECTIVES: Machine must substantially improve on the approximately 1% down time of present version. Machine should run 24 hours a day year-round.
2.6. DYNAMIC VARIABILITY: We hope that the design will embody soft failures wherein bandwidth capability will degrade with failure but no functions will be totally lost.
2.7. PENALTIES: Reduction in communication facilities to a net. Multiple failures can cause loss of communication to certain nodes.
2.8. CONSTRAINTS: No explicit constraints--goal is to have a few racks in size and cost of about $100,000.
2.9. TRADEOFFS: Cost and everything else.

3. DESCRIPTION
3.1.1.1. INTERCONNECTIVITY: See Figure.
3.1.1.2. RANGE: Smallest is single processor single bus system with a single logical memory. We do not understand maximum size constraint as a number of physical and engineering problems (power, cooling, rack space, cabling) limit the size before logical boundaries are reached. We are building a 14 processor prototype and expect that systems of twice that size are not much harder.

3.1.1.3. CAPABILITY: That of a single Lockheed SUE (a small modern 16 bit mini) 200,000 Adds/sec

3.1.2 EXECUTIVE
3.1.2.1. MODES: Designed for parallel task execution of specially coded real-time problems. Parallelism is not decided upon in advance but is provided for. All processors can perform all tasks and adjust to current work load.

3.1.2.2. SOFTWARE: Split into tiny (300 microsecond) tasks which are queued with the aid of special hardware (which itself is replicated for reliability). All processors can perform all jobs.

3.2. FAULT TOLERANCE
3.2.1. FAULTS TOLERATED: We believe that, short of system power failure, any one piece of the system can fail without loss of function but with loss of bandwidth capability.

3.2.2. FAULTS NOT TOLERATED: Malicious manual interference, systemic power failure, etc. Little protection against software faults included since the program is a dedicated real time program not subject to the vagaries of "users".

3.2.3. TECHNIQUES: Redundancy of parts and nonspecialization of processors. Parts connected in a network so that communication paths don't force specialization, e.g., I/O devices connect to two busses--each of which can be reached by any of k processors. Power is distributed as 110 AC and power supplies are modular--i.e., each unit has its own DC supply with it--also its own cooling. The system requires each piece to perform certain tests periodically and one of the tasks required of some randomly selected free processor is to check up on how everyone else is doing. Modules can disconnect one another from the system if failing operation is detected but protection is provided to avoid inadvertent decoupling of a good unit.

3.3. NOVELTY: We do not know of a similar system of a collection of task oriented "workers" sharing responsibility not only for the routine workloads (with variations) but also for self test and, if appropriate, amputation.

3.4. INFLUENCES: Macromodular project at Washington Univ.
3.5. HARD-CORE: We have tried to avoid this concept in our system wherever we could. We hope that it is in this very avoidance that we may improve reliability (see 3.2.1).

4. JUSTIFICATION
4.1. RELIABILITY EVALUATION: We believe that in a new system of this sort it is difficult if not impossible to make meaningful prognostications of reliability. We believe our overall system design is prone to reliability if the basic parts are themselves reasonably reliable. Only after the system has been running for a year or two will we begin to understand what its real reliability is.

4.2. COMPLETENESS OF EVALUATION: Not particularly with regard to part failures; conceptually, we believe it is quite complete.

4.3. OVERHEAD: Impossible to estimate since this was not a primary goal at the outset. The original goal was high bandwidth and the scheme we chose simply led naturally to what seemed a very reliable looking structure. We have added relatively little (10%) specifically for reliability. We could add more and (hopefully) improve the reliability more as our structure is modular and expandable.

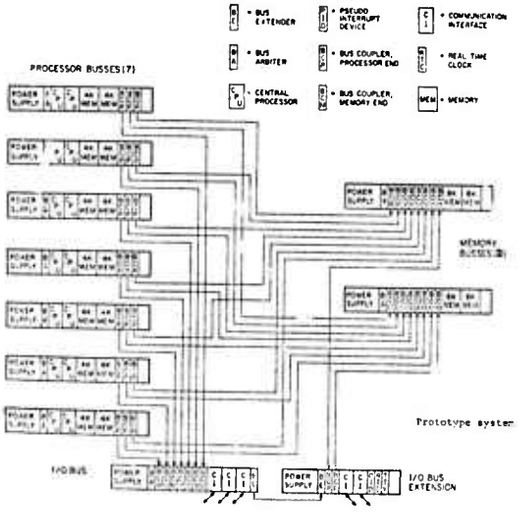
4.4. APPLICABILITY: We have only begun to investigate these possibilities. We hope there will be many. Among these we see real time signal processing and some specialized multi-user applications.

4.5. EXTENDABILITY: The system is designed to be generally extendable. That is one of its main points. Certain boundaries are reached where the next step in expansion is more costly than prior steps. We do not know where hard limits will appear. We believe they will, for some time, be of an "engineering" rather than "logical" nature.

4.6. CRITICALITIES: Fairly well matched. Since goal was for variability the question is not too meaningful. Multiprocessing was not a goal; it was, for us, a means. Hardware choice was for suitability and convenience.

4.7. IMPLICATIONS: At present the design is based on the Lockheed SUE bus structure (slightly modified). It could have been based on some other computer, but less easily and at greater cost. Until or unless we switch, this means that all units in the system follow the SUE bus discipline. The overall design was conceived for problems that could be broken conveniently into parallel executable tiny tasks. It achieves speed and power by such parallelism.

5. CONCLUSIONS
5.1. STATUS: Prototype nearing completion.
5.2. EXPERIENCE: It is hard to build such systems.
5.3. FUTURE: This IMP will be incorporated into the ARPA network in various sized configurations.



SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS

W. C. Carter
IBM Thomas J. Watson Research Center
Yorktown Heights NY 10598

1. IDENTIFICATION
1.1. NAME: I am reporting mainly on a long-term research effort in techniques for fault-tolerant computer architecture. The relevant prior publications have used, for example, the terms "modular architecture", "self-repairing computers", "dynamic checking", "fault diagnosis", "stand-by sparing" or "dynamic recovery" in the titles and the authors have been some subset of the participants named in 1.4. For present purposes I will talk about a paper Modular Digital Computer system called MDC whose principal properties will be specified later. For reality, some requirements will be imposed which have nothing to do with fault tolerance per se. This system does not really exist, and will not exist, but is specified to provide a focus for our fault tolerant computing research.

1.2 RESPONSIBILITY: IBM Research.

1.3 SUPPORT: Support has come from IBM, U. S. Air Force and NASA.

1.4 PARTICIPANTS: W. G. Bouricius, W. C. Carter, E. P. Hsieh, D. C. Jessep, Jr., G. P. Putzolu, J. P. Roth, P. R. Schneider, C. J. Tan, A. B. Wadia.

1.5 START: Formal initiation occurred in March, 1966.

1.6 COMPLETION Open ended. No end item is scheduled.

1.7 BIBLIOGRAPHY:
*Roth, J. P. "Diagnosis of automata failures: a calculus and a method", IBM Journal,vol. 10, 4, 1966.

*Bouricius, W. G., Hsieh, E. P., Putzolu, G. R., Roth, J.P., Schneider, P. R., Tan, C. J., "Algorithms for detection of faults in logic circuits", IEEE TC, Vol. C-20, Nov. 1971.

*Bouricius, W. G., Carter, W. C. and Schneider, P. R., "Reliability modeling techniques and tradeoff studies for self-repairing computers", ACM National Conference, San Francisco, California, August, 1969.

*Bouricius, W. G., Carter, W. C., Roth, J. P. and Schneider, P. R., "Investigations in the design of an automatically repaired computer", Paper Number 6.4 Conference Digest of the First Annual IEEE Computer Conference, Chicago, Illinois, September 6-8, 1968.

*Carter, W. C. and Schneider, P. R., "Design of dynamically checked computers", IFIPS, Edinburg, Scotland, August, 1968.

*Carter, W. C., Jessep, D. C., Wadia, A. B., "Error-free decoding for failure-tolerant memories", 1970 IEEE Computer Conference, Washington, D. C., June, 1970, pp. 229-239.

*Carter, W. C.,Jessep, D. C., Bouricius, W. G., Wadia, A. B., McCarthy, C. E., Milligan, F. G., "Design techniques for MARCS" (Modular Architecture for Reliable Computer Systems), NASA Contract NAS8-24883, RA12, IBM T. J. Watson Research Center, Report Number 70-208-002, March 26, 1970.

*Carter, W. C., Jeasep, D. C., Wadia, A. B., Schneider, P. R., Bouricius, W. C., "Logic design for dynamic and interactive recovery", IEEE TC, Vol. C-20, Nov. 1971.

2. MOTIVATION
2.1 PURPOSE: Real time control, data acquisition and data management.

2.2 PHYSICAL ENVIRONMENT: Aerospace applications have predominated in specific design decisions. Modularity should ensure wide applicability.

2.3 COMPUTING ENVIRONMENT: The MDC is planned to be able to run the gamut from being insulated from human control, serving a variety of sensors and effectors, to being able to accept ground-based human directed control.

2.4 COMPUTING OBJECTIVES: Predicted configuration scaleability primarily under internal control including systems which are fault tolerant by masking redundancy, by stand-by redundancy, or by software checks; systems whose use of power is variable (but whose thruput is affected); and systems operating in parallel. The major objective is to provide means for meeting various requirements with a high degree of confidence.

2.5 RELIABILITY OBJECTIVES: The system is to be designed to meet varying specific mission reliability objectives with a high degree of certainty. Examples are survival for n years with a probability p; "fail operational, fail operational, fail safe", or reliability variable with mission task.

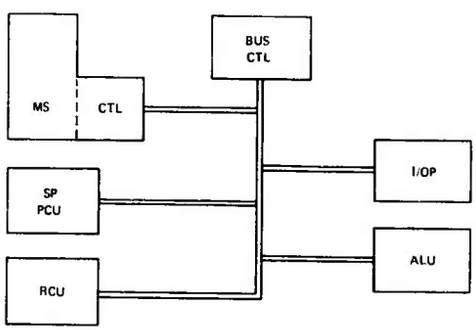
2.6 DYNAMIC VARIABILITY: As stated above, dynamic variation of system parameters such as performance, reliability and power consumption with confidence in the design as a major objective.

2.7 PENALTIES: Variable with mission, ranging from loss of human life through expensive flight hardware to abortion of flight objectives.

2.8 CONSTRAINTS: Hardware must be designed to fit weight, power and size requirements, yet able to have thruput compatible with mission requirements and to support the software necessary for reasonable programming effort per mission.

2.9 TRADEOFFS: Hardware efficiency and potential thruput are traded for 1) system reliability as defined per mission phase; 2) simplification of recovery process and other basic executive functions; 3) high malfunction coverage and design certification; 4) ease of program validation; 5) convenience of programming and ease of diagnosis for external equipment; 6) system flexibility.

3. DESCRIPTION
3.1 ARCHITECTURE
3.1.1. CONFIGURATIONS
3.1.1.1. INTERCONNECTIVITY: The basic uniprocessor configuration consists of partitioned computer subunits attached to several busses. The basic subunits are (see attached rough diagram): ALU, Scatch and Program Control Unit, Bus Control, I/O Processor and Recovery Control Unit. The bus orientation remains, but the units may be modified (microprogrammed) for varying missions. The system consists of replicas of the basic subunits, with configuration control governed by the RCU and Executive Program. A major problem is the interface design to meet the constraints of fault tolerance, long life, and varying modes of operation. The memory is encoded with a b-adjacent error correcting code and spare b wide subunits per basic module.



3.1.1.2. RANGE: The range of the system is not frozen in the architectural concept. After four processors the law of diminishing returns sets in sharply and further partitioning may well be a better bet for long life. The memory will consist of modules, each module consisting of b-wide units with b-adjacent coding and spare b-width units. The upper limit depends upon the hardware available, but hardware does not appear to be critical.

3.1.1.3. CAPABILITY: The order of 10E5 to 10E6 additions per second per basic system with a minimum of 256K-512K 32-bit words of memory. I/O will be handled by up to 4 16-bit parallel channels with 50,000 transfers per second simultaneously on one input and one output channel. The I/O processor will handle the details of I/O control under direction from the processor Executuve.

3.1.2 EXECUTIVE: The standard executive control (allocation, scheduling, dispatching, I/O) will be achieved by replicated software routines. These tasks have not been studied much.

3.1.2.1. MODES OF OPERATION: Each processor is multiprogrammable. System operation includes fault masking, multiprocessing with hardware fault detection and multiprocessing with software analysis. The mode of operation of most concern is that of recovery initiation, the interaction of the recovery and error analysis programs of the executive and the RCU. Recovery and audit programs always run background whether the system is in fault masking, fault detection or software analysis modes.

3.1.2.2. SOFTWARE ORGANIZATION: The system software will be distributed among the processors and analyzed by audit routines for early detection of errors.

3.2. FAULT TOLERANCE

3.2.1 FAULTS TOLERATED: In the error-masking mode, any number of faults which affect only one partitioned sub-unit can be tolerated. The system handles transient faults with instruction retry or permanent faults with hardware controlled reconfiguration. The cause is irrelevent as long as the interface detects disagreement. The disagreement circuits are self-checking so faults in them are detected. Initially the same malfunction in three units is necessary to defect the system. After reconfigurations two faulty units may escape detection. In the error detection mode, faults causing a single subunit to be in error are detected. At this point the same errors in two units will be undetected. Diagnosis and software recovery is necessary for continuation.

Faults detected by software checks are detected and recovery should follow in the unchecked multiprocessing mode. Faulty software may be detected by the RCU time-out tests and system evaluation procedures.

3.2.2. TECHNIQUES: In hardware fault tolerant mode the system should FO - FO - FS for each one of the partitions of the system if four copies of the basic computer are used. Diagnosis can continue the computation with one partition unchecked. Detailed fault analysis must be performed to validate such goals. In hardware fault detection mode the system should run at least two multiprocessor hardware checked systems. A fault would be detected, and diagnosed would allow continuation with one partition unchecked by hardware. Achieving such hardware/firmware/diagnosis goals depends upon the development of many tools of fault analysis. The memory encoding is b-adjacent multiple error correcting and/or multiple b-adjacent error detecting. The codes used are variants of Reed-Solomon codes with combinational self-checking translators which pass only correct code words. Standard single instruction retry is available.

Microdiagnostics under executive program control with program variable input patterns will be used for fault analysis. The executive software will use the standard fault tolerant techniques - two way lists with pointer verification before proceeding, stored data and programs will be tagged with redundant identification, read only programs will allow simple updating etc. Rollback and restart will be used for multi-processing with hardware or software error detection. The RCU monitors constantly for catastrophic faults - those not detected by the hardware and software tests. The standard time-out tests and system performance evaluation routines are run and controlled by the RCU. Power is conserved under program control by forcing n cycles between memory accesses, imposed by a counter with program changeable contents.

3.3 NOVELTY: Reconfiguration under hardware control in fault masking mode. Choice of computer fault masking, multiprocessing with fault masking and various forms of detection, multiprocessing with hardware error detection by comparison, multiprocessing with software error detection. Storage reliability by b-adjacent multiple error detecting and correcting codes. Self checking memory translators, checking circuits, and error-analysis ciruits. Use of power under program control.

3.4 INFLUENCES: 1. JPL Star - the total effort; 2. SRI, Techniques for the Realization of Ultra-Reliable Spaceborne Computers; 3. MIT -Draper Lab. for spaceborne multiprocessors; 4. Rapid emergence of LSI for feasibility of much redundant hardware.

3.5 HARD-CORE: Assuming that hard core means hardware, redundant or not, whose failure will produce undetected errors, there is no such hardware in this system. Hopefully, the software can be validated so that equal claims can be made for it.

4. JUSTIFICATION FOR THE SYSTEM

4.1 RELIABILITY EVALUATION: Architectural reliability evaluation by interactive program using exponential failure assumption for the units. Determination of component failure rates by analysis based upon previous data, experience, and analysis. Logic fault analysis of circuits in design stage by interactive fault simulation programs. Diagnostic pattern evaluation by simulation programs. Memory failure predictions by careful probabilistic fault analysis to predict error patterns, programmed computation of the circuit failure constants, programmed evalutation of reliability. Programmed analysis of RCU functions. Theoretical analysis of design, with hardware and software, in complicated situations (guided by simulation).

4.2 COMPLETENESS OF EVALUATION: Major unsolved problem.

4.3 OVERHEAD: Variable. In the processors about a 3 1/2 :1 logic count penalty is paid (the cost is much less). In the memory about a 3:2 storage penalty is paid. In the software the cost is unknown, but considerable.

4.4 APPLICABILITY: The concepts can be used elsewhere, the system is oriented toward space and extremely high reliability applications.

4.5 EXTENDABILITY: This computer is too reliable to fit into most other systems. For extension some of the fault tolerent techniques in the computer must be eased for better total system balance.

4.6 CRITICALITIES: Multitasking, as with all Executive-controlled recovery systems, is critical, achieved here with multiprogramming. Multiprocessing is an imposed condition, but small system simplifications would result if this condition were relaxed. Design validation tools are critical.

4.7 IMPLICATIONS: Architects must perform automated error and recovery analysis while doing system specification. Human analysis is too fallible. Hardware designers must have and use tools to do fault analysis as they design. After the first pass they must do design validation and iterate. Software designers must participate in the initial decisions, must produce more techniques for producing self-checking programs, and must produce the tools for program validation. Applications programmers must validate their programs (top down programming techniques will help), and must follow system rules (not so far known).

5. CONCLUSIONS

5.1 STATUS: This system is the collection of a group of ideas from a research project.

5.2 EXPERIENCE: None to report to date.

5.3 FUTURE: The system will be pursued only in a modified form as a paper study only.

5.4 ADVANCES: The problems of validation - hardware and software - will provide many a bottleneck for fault tolerant computing. The basic problem of definition of fault tolerent computing will be with us - do we consider any algorithm, procedure?

SURVEY ON FAULT-TOLERANT COMPUTER SYSTEMS

Albert L. Hopkins, Jr., MIT Draper Laboratory
Cambridge, Mass. 02139, Feb 1973

1. IDENTIFICATION

1.1. NAME: I am reporting on a long-term development effort which has been supported by different projects at different times. The following titles have been used for published reports:

* "A Fault-Tolerant Information Processing System for Advanced Control, Guidance,and Navigation".

* "Space Transportation System Data Management System".

In addition, an experimental three-processor three-scratchpad breadboard has been given the acronym CERBERUS for the three-headed dog in classical mythology. The acronym engendered the title: Controlled Error Recovery Behavior Employing Redundant Use of Scratchpads. In what follows, I use "the system" to mean the general concept, rather than a specific hardware design.

1.2. RESPONSIBILITY: This work is in the Digital Development Group of the Charles Stark Draper Laboratory, a division of M.I.T.

1.3. SUPPORT SOURCES: So far all support has come from the NASA Manned Spacecraft Center.

1.4. PARTICIPANTS: MIT and NASA/MSC.

1.5. START: Work in this area began in 1966.

1.6. COMPLETION: Open ended. No end item is scheduled.

1.7. BIBLIOGRAPHY:

* R. L. Alonso, A. L. Hopkins, Jr., and H. A. Thaler, "Design Criteria for a Spacecraft Computer", Spaceborne Multiprocessing Seminar, pp. 23-28, NASA ERC, Boston Museum of Science, Oct. 1966.

* R. L. Alonso, A. L. Hopkins, Jr., and H. A. Thaler, "A Multiprocessing Structure", Digest of the First Annual IEEE Computer Conf., pp. 56-59, Chicago, Sept. 1967.

* A. I. Green et al., "STS Data Management System Design", MIT C.S. Draper Laboratory, Cambridge, Mass., Report E-2529, June 1970.

* A. L. Hopkins, Jr., "A Fault-Tolerant Information Processing Concept for Space Vehicles", IEEE Trans. Computers, Vol. C-20, pp. 1394-1403, Nov. 1971.

2. MOTIVATION

2.1. PURPOSE: Real time control, data acquisition and data management.

2.2. PHYSICAL ENVIRONMENT: In principle it could be any, but aerospace applications have predominated in design decisions.

2.3. COMPUTING ENVIRONMENT: Systems considered here are envisioned as largely self-contained information processing systems serving a variety of sensors and effectors including human operators. Such systems would be distributed, hierarchical and redundant. Central fault-tolerant multiprocessors would communicate over serial data buses to local processor complexes embedded in subsystems of the total system. A principal application considered for this approach was the Space Shuttle, where the Orbiter would have one central multiprocessor with adequate redundancy and spare hardware to be operational after three malfunctions. Each subsystem or group of identical subsystems would be served by simple or redundant local processors, as appropriate, to fulfill the redundancy requirement for that subsystem or group.

The Booster stage of the Space Shuttle would, in this concept, contain a system similar to that of the Orbiter, capable of communicating with it by way of a serial bus connecting the two central multiprocessors. All communication between a central multiprocessor and its local processors would be via a serial data bus.

2.4. COMPUTING OBJECTIVES FOR THE CENTRAL MULTIPROCESSOR: Variable from the order of 10E5 (i.e., 10 to the 5) to the order of 10E6 operations per second, with memory capacities of from 2E14 to 2E17 words of main random access memory. Input-output bandwidth 10E5 useful bits/sec on a 1CE6 pulse-per-second bus. Reaction time order of 10 milliseconds.

2.5. RELIABILITY OBJECTIVES: Various types of objectives. One example is airline applications where less than one catastrophic system malfunction in 10E7 flights is sought. Other objectives are stated in terms of the number of individual malfunctions which can be tolerated in a flight, such as "Fail operational, fail operational, failsafe" (FO-FO-FS). The system is generally meant to be used in very high reliability applications.

2.6. DYNAMIC VARIABILITY: Graceful degradation is available as a means of exchanging performance for reliability.

2.7. PENALTIES: In the Space Shuttle application, as in possible aircraft applications, human life is concerned. Other life-critical applications can be easily envisioned.

2.8. CONSTRAINTS: In Space Shuttle and aircraft, approximately 2 cubic feet, 120 lb., 300 watts. (Estimate for a central multiprocessor). Other applications may be more or less severe.

2.9. TRADEOFFS: Hardware efficiency is traded for 1) system reliability, 2) high malfunction coverage, 3) ease of program verification, 4) system flexibility.

The number of faults tolerated is variable through a combination of replication and sparing. Processors and memories can be added (deleted) to increase (decrease) processing and memory resources.

3. DESCRIPTION OF THE SYSTEM

3.1. ARCHITECTURE

3.1.1. CONFIGURATIONS

3.1.1.1. INTERCONNECTIVITY: The system makes extensive use of replication, and consequently connections have a high cost. Serial and byte-serial buses are used between basic units. Multiplexers are employed to prevent single unit malfunctions from spreading to all copies of a redundant bus. The canonical interconnection scheme is shown in Figure 1.

3.1.1.2. RANGE: No range limits have been determined, but the following numbers may be typical for an aerospace application. There are two current competitive conceptualizations of the system. These numbers represent the newer and less well developed concept.

* 6= Number of simultaneous job steps in process
* 3= Degree of replication of each processor-scratchpad
* 3= Number of spare processor-scratchpads
*21= Total processor scratchpads = 6 x 3 + 3
* 4= Number of independent memory blocks of 16K
* 3= Degree of replication of each block
* 3= Number of spare blocks
*15= Total memory block modules = 4 x 3 + 3

The number of processor-scratchpads and memory blocks can be increased up to the practical bandwidth limit of the processor-memory bus and the I/O bus.

3.1.1.3. CAPABILITY: The order of 10E5 to 10E6 additions per second and the order of 2E14 words of memory. Three processors would be the smallest "sensible" number.

3.1.2. EXECUTIVE

3.1.2.1. MODES OF OPERATION: All programs are segmented into job steps which are dispatched by a floating form of executive. Each job step occupies one processor full time while it runs. Multiprocessing is the normal operating mode. Multiprogramming of each processor is not envisioned.

3.1.2.2. SOFTWARE ORGANIZATION: I/O processing is quasi-dedicated to one processor triplet (i.e. it can float but does so only when malfunction makes it necessary). Executive, monitor, and reconfiguration programs are run on an as-needed basis by each processor triplet as it finishes a job step.

3.2. FAULT TOLERANCE

3.2.1. FAULTS TOLERATED: Individual units (e.g. processor, memory unit, multiplexer) can malfunction one at a time with no restriction on what the nature of the malfunction is. Errors are masked by the system until it reconfigures itself to a fault-tolerant state.

3.2.2. FAULTS NOT TOLERATED: Certain malfunction pairs which occur simultaneously or close together in time can produce loss of data and many require a program restart. Incorrect specifications or program malfunctions can defeat the system. Systematic hardware malfunctions in which the same malfunction occurs in two redundant units can defeat the system.

3.2.3. TECHNIQUES: Two different concepts.

First concept: all processors are duplexed for detection. All scratchpads are triplexed for masked dump capability. Single instruction restart. Graceful degradation of processor-scratchpad groups. Triplex memory units with dedicated spares. Triplex buses with spares. Multiplexers isolate buses from failed groups of units.

Second concept: processor-scratchpad units are organized into groups of three under software control. Each looks for disagreement. If disagreement occurs, continue running to end of job step, then enter reconfiguration program. Graceful degradation of individual processor-scratchpad units (rather than groups of three scratchpads and two processors as in first concept). Triplex memory units with non-dedicated spares. Triplex buses with spares. Multiplexers isolate buses from failed individual units (rather than groups as in first concept).

In both concepts, software configuration control is used, which is valid as long as a working processor group, memory group, and bus-multiplexer group are available. Multiplexers participate in configuration control.

3.3. NOVELTY: Single instruction restart. Absence of interrupts and program rollbacks. Distributed monitor and reconfiguration functions. Use of multiplexers to isolate bus and unit malfunctions. Fault-tolerant clock. Hierarchical system with fault tolerance extended into subsystems.

3.4. INFLUENCES: Rapid emergence of LSI memories and processors has encouraged use of replication and partitioning with simple, identical units. Apollo Guidance Computer experience prompted elimination of interrupts and rollback for the sake of program verification. Carter and Bouricius for reliability models. Avizienis for concepts of fault tolerance.

3.5. HARD CORE: Assuming that hard core means non-redundant hardware, there is no hard core in this system. Configuration control is a software function using the available hardware to configure the system.

4. JUSTIFICATION

4.1. RELIABILITY EVALUATION: So far mostly geared toward FO-FO-FS. Some Probabilistic analysis. No reliability projections as yet since hardware has not been selected and failure rates are therefore not known.

4.2. COMPLETENESS OF EVALUATION: Hardware not selected, hence failure rate not known.

4.3. OVERHEAD: About 80% of the system is devoted to the achievement of fault tolerance.

4.4. APPLICABILITY: This concept is applicable to most digital control environments, depending on the economics of the application regarding fault tolerance.

4.5. EXTENDABILITY: Extendability probably does not apply, since the system is still loosely specified.

4.6. CRITICALITIES: The system is most cost-effective compared to other systems when the number of faults to be tolerated is high and where ultra-high reliability is sought. For single-fault tolerance and less high reliability, the system configuration might be changed.

4.7. IMPLICATIONS: In an ultra-high reliability application, specifications and programs must be proven to be correct. In this system, applications programmers must also segment their programs into short job steps.

5. CONCLUSIONS

5.1.1 STATUS: This is a research project with a breadboard experimental unit almost completed.

5.2. EXPERIENCE: None to report to date.

5.3. FUTURE: Some parts of the system still need to be designed and prototyped. Experiments must be conducted on a full-scale prototype system.

5.4. ADVANCES: The following will be beneficial.

*Demonstrated field experience with various fault-tolerant concepts.

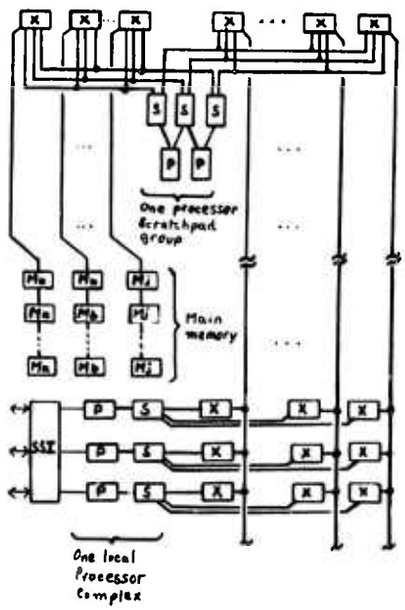
*Practical techniques for generating correct programs.

*Practical ways of verifying that a program is correct.

6. COMMENTS

The questionnaire was good in the sense of being thorough, but in my haste to respond to it I wonder if I have omitted significant material. An additional comment about this system is that it has been configured around integrated processors and memories which resemble those that are available today. The hardware efficiency number given in Section 4.3 is very misleading, because the cost of the hardware can be the least important cost of the system, if the hardware is conventional and not overly expensive. This system is expected to save in costs of system integration, program verification, and operational reliability experience. These savings may be far in excess of the hardware cost.

As an additional note, the replicated approach used here gives coverage of 1.0 for single malfunctions. Most coded approaches generally give lower coverage, difficult to quantify, and often impossible to verify in the field.



P...Processor
S...Scratchpad memory
M...Memory module
X...Multiplexor
SSI...Subsystem interface

SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS

W. L. Martin, Hughes Aircraft Company
Fullerton, California 92634, May, 1972

1. IDENTIFICATION
1.1. Name: Automatically Reconfigurable Modular Multiprocessor Systems (ARMMS)

1.2. RESPONSIBILITY: Astrionics Laboratory, Marshall Space Flight Center, NASA

1.3. SUPPORT: Same as 1.2

1.4. PARTICIPANTS: The participating organizations include NASA MSFC, Hughes (system design), M&S Computing, Inc. (executive software under subcontract to Hughes); Auburn University (executive control approaches under contract to NASA). Principal participants by name are as follows: NASA - Dr. J. B. White, Sherman Jobe; Hughes - W. L. Martin; M&S - T. T. Schansman; Auburn - Dr. David Irwin.

1.5. START: The date of conception was circa 1968 in a concept document written by Dr. White. MSFC has been developing technology under their Space Ultrareliable Modular Computer (SUMC) program since shortly thereafter. The system design effort being performed by Hughes was solicited in May, 1971 with a contract in October, 1971.

1.6. COMPLETION: The Hughes system definition contract will be completed in April, 1973. Construction of a breadboard or prototype may follow with completion date uncertain.

1.7. BIBLIOGRAPHY: Various planning documents have been written at NASA. Dr. White may be contacted for these. The Hughes effort is divided into three phases, with the Phase 1 report released on April 15, 1972. It is titled "Design of a Modular Digital Computer System", DRL 4, Phase 1 Report, Hughes Aircraft Company FR 72-11-450. Two other papers have been submitted for publication. Their fate is uncertain as of yet, but interested parties may obtain copies from W. L. Martin at Hughes. These are the following:

*J. L. Bricker, A Unified Method for Analyzing Mission Profile Reliability for Standby and Multiple Modular Redundant Computing Systems which allows for Degraded Performance (submitted to the IEEE Transactions on Reliability Theory).

*J. L. Bricker and W. L. Martin, Reliability of Modular Computer Systems with Varying Configuration and Load Requirements (submitted to 1972 IEEE Computer Society Conference).

2. MOTIVATION
2.1. PURPOSE: ARMMS is to be applicable through modularity to diverse types of space missions ranging from launch vehicles, to space stations to deep space probes.

2.2. PHYSICAL ENVIRONMENT: Spaceborne

2.3. COMPUTING ENVIRONMENT: See 2.1, 2.2.

2.4. COMPUTING OBJECTIVES: The motivating computing objective is to be able to configure systems which are fault tolerant through TMR or other redundant modes or to use the modules in parallel for high computing capacity and to be able to reconfigure from one type to the other dynamically. Maximum capacity in a non-redundant mode is to be "several million" additions per second.

2.5. RELIABILITY OBJECTIVES: One specific reliability objective is that the probability of survival of at least a simplex computer after 5 years should be at least 0.9 (with no on-board maintenance). The overall intent, however, is that the system should be able to be configured to meet specific mission reliability objectives whether they be stated in terms of maximum recovery time, number of failures tolerated, etc.

2.6. DYNAMIC VARIABILITY: As noted in 2.4, dynamic variability of configuration is one of the primary motivations.

2.7. PENALTIES: See 2.1.

2.8. PHYSICAL CONSTRAINTS: There are no explicit physical constraints except those implied by the nature of the intended spaceborne application. However, an implicit physical constraint is the difficulty of contriving an approach to a large (by aerospace standards) computing capability fault-tolerant design within the confines of weight and power budgets which may prevail for interplanetary missions.

2.9. TRADEOFFS: At the current stage of the design, there are many critical tradeoffs yet to be made.

* For a computer which will be built after 1975, what device complexity and failure rates should be assumed? Almost all aspects of the design are critically affected by this question. Some of the more crucial ones are the maximum complexity of any module; the degree to which processors must be sub-partitioned; the resulting cost in switching hardware; the maximum number of replicates of any one module type which must be accommodated; and the complexity of the configuration control software.

* The basic ARMMS concept developed by NASA incorporates a dedicated executive module rather than a floating executive. Resulting tradeoffs include specific definition of functions to be performed, specification of status monitoring and reconfiguration parameters, and a design approach which yields sufficiently high reliability for the executive module.

* The system architecture is not yet defined in any complete sense. Questions yet to be resolved include specific definition of allowed modes of operation; definition of the means of interconnecting the modules; placement and use of voters; use of error-correcting codes for memory data; maximum number of replicates per module class; specific techniques for memory data protection; and fault tolerance features within each module class. At present, we are making tradeoffs based on two major configuration alternatives. Although few tradeoff conclusions have been reached, the predominating evaluation criteria are almost certain to be the following:

* Implementation feasibility - Any design feature which does not seem to us to be feasible in any major sense (e.g., pin count, excessive power, design cost) will be rejected. We are not particularly interested in developing new theories or techniques of fault-tolerant computing but are very interested in developing a much-needed testbed based on the research performed over the last 5-10 years.

* Suitability to the multi-mode configuration requirements - ARMMS is intended to be usable in configurations ranging from a simplex computer to TMR with standby spares. Any feature which imposes excessive overhead cost for the benefit of one configuration at the expense of others is suspect. For example, added hardware per module for internal fault tolerance multiplies the hardware penalty paid in TMR mode.

3. SYSTEM DESCRIPTION: As seen from the tradeoff discussion above, no firm system description is possible now. Therefore, the responses in this section are necessarily brief and incomplete.

3.1. ARCHITECTURE
3.1.1. CONFIGURATION
3.1.1.1. INTERCONNECTIVITY: All processors, I/O, and executive controller may access all of main memory (a study of the desirability of identifying an additional level of memory, cache or task oriented was made, with a negative conclusion reached). The most probable scheme is a system of replicated busses with access control governed by the executive module. The nature of spaceborne I/O activity is biasing us toward a direct processor I/O data path which can be used for transmitting short bursts of data. The executive controller will monitor the other system modules via a time-shared bus. This bus ordinarily polls the modules in sequence but may be interrupted by the processors on task completion or other time-critical event. No direct interaction of modules of a given class (e.g., processor-to-processor) is planned.

3.1.1.2. RANGE: The general approach to achieving the large capacity mentioned previously is to maximize the individual processor performance so that throughput is not dependent on a large number of parallel instruction streams. (Three is a desirable upper limit.) The maximum main memory capacity is to be large enough (e.g., 256K-512K words) to support the high-throughput goals. The word length is to be 32 bits as dictated by the choice for the NASA SUMC processor. Cumulative I/O data rate capability is to be 10 million bits per second. In all cases, maximum number of modules per class (and the memory module capacity) will be determined primarily by reliability considerations. A least upper bound is 4 for each class.

3.1.1.3. CAPABILITY: (See 2.4.)

3.2. FAULT TOLERANCE: (The system is still too much conceptual to allow a decent response. All faults are to be tolerated. None are to be not tolerated. All techniques will be considered. Ask again in a year and let's see what turned out.)

3.3. NOVELTY: On the one hand, there's nothing that one can point out as being fundamentally novel (this is true of most machines, I think). On the other hand, there are no machines that I know of that have successfully implemented a variable redundancy approach such as is being sought. The choice of a dedicated executive module is the only deviation at the block diagram level from other multiprocessors (but this module is a rather close parallel of the TARP in STAR).

3.4. INFLUENCES: JPL STAR; NASA ERC Modular Computer; NASA MSFC SUMC; IBM, "Architectural Study for a Self-Repairing Computer; SRI, Techniques for the Realization of Ultra-Reliable Spaceborne Computers.

3.5. HARD-CORE: The executive module is hard-core. The effect is to be minimized by simplifying the module as much as possible and by internal redundancy (which may ultimately result in replication).

4. JUSTIFICATION
4.1. RELIABILITY EVALUATION: To date, reliability has been evaluated solely by analysis (as described in the two papers mentioned in 1.7). Later in the effort, we expect to extend the analysis to include coverage and switch unreliability. We also expect to simulate the logical performance of the intermodule switches and to simulate the injection of faults.

4.2. COMPLETENESS OF EVALUATION: I'm not sure that I understand the question. But whatever you mean by design evaluation, I'm sure that I wish we had more time and money to do it better.

4.3. OVERHEAD: Since the configuration is dynamic, the percentages of resources attributed to the achievement of fault-tolerance also vary with time. An upper limit is probably 80%; a lower limit is probably 20% (in cost, logic, execution time, etc.).

4.4. APPLICABILITY: Applicability to other than space applications is questionable.

4.5. EXTENDABILITY: I think that it is more likely that the system design can usefully contract than that it can be usefully extended.

4.6. CRITICALITIES: The major difficulty of the design is the breadth of the goals. The critical problem is therefore to find a set of design choices which complies reasonably well with all the goals (e.g., we want high speed and capability but require low weight and power). However, I don't think that slight changes would critically affect the design. (Also, as a side observation, while one is in the midst of a system design, all choices seem critical, don't they?)

4.7. IMPLICATIONS: (Let me plead that this question seems too vague. I don't know where to start with a brief response.)

5. CONCLUSIONS
5.1.1 STATUS: The status is sufficiently described by the above comments, I think. In summation, we are about one-third of the way through a system definition phase.

5.2. EXPERIENCE: It appears that component technology is contributing more to the feasibility of highly reliable machines than architecture concepts are. As recently as 2 or 3 years ago, gate failure rate of 10E-7 per hour seemed optimistic. At present, gate failure rates of 10E-10 per hour are credible for the space environment. On the other hand, the assumption that dormant failure rates are a small fraction of active failure rates appears questionable. For a long-life machine in an unmanned environment, these two factors are of major significance to the system designer.

5.3. FUTURE: There are two conflicting possible futures of ARMS. The pessimistic view is that it will go the way of 10 or 15 similar paper design efforts and will die with only a final report to commemorate its non-existence. The optimistic view is that it will appear sufficiently promising in concept that NASA will continue its development and eventually attach it to a mission. Planning is of course being directed toward the optimistic alternative.

5.4. ADVANCES: I cannot add anything to the lists of theoretical problem areas and needed areas of investigation which SRI described in its reports under contract NAS 12-33. In particular, I agree that there have been too few case studies which can be evaluated.

A major practical advance which is needed is the identification and exploitation of specific applications in which fault-tolerant machines can be justified economically. It is significant, I think, that the Bell ESS-1 and System-360 FLT's instruction retry, etc., represent the most extensive application of fault-tolerance and diagnostic techniques. Both are in areas where the payoff for high reliability is great. Although aerospace applications have supported much of the research in fault-tolerant machines, I am skeptical that there is a sufficient mass of money there to lead to very widespread results in fielded systems. The situation is analogous to that which has existed for associative processing for 10 years, in that the glamour, concepts, and techniques are often apparent but cost considerations ultimately lead to more conventional choices.

Also, I wonder if "fault-tolerant computing" is too narrow a view and that many of the basic ideas would be applicable to a discipline of "Fault-tolerant systems". Perhaps there are other equally fertile, but less plowed, fields to be conquered.

6. COMMENTS: (See 5.4)

SURVEY OF FAULT-TOLERANT COMPUTER SYSTEMS

James S. Miller, Intermetrics, Inc., 701 Concord Avenue, Cambridge, Massachusetts 02138, March 1973.

1. IDENTIFICATION

1.1. NAME: The system is referred to as either the Intermetrics Multiprocessor, or the Space Station Computer.

1.2. RESPONSIBILITY: Intermetrics, Inc.

1.3. SUPPORT: NASA Manned Spacecraft Center, Houston, Tex.

1.4. PARTICIPANTS: J. S. Miller, W. H. Vandever, S. F. Stanten, A. E. Avakian, and A. L. Koemala.

1.5. START: The project began in June, 1969, and continued for ten months. After a thirteen-month period of inactivity, the design effort was resumed in May, 1971.

1.6. COMPLETION: The second phase of the design was completed, and a report published, in September, 1972.

1.7. BIBLIOGRAPHY:

[1]J. S. Miller, D. J. Lickly, A. L. Koemala, and J. A. Saponaro, "Final Report--Multiprocessor Computer System Study", Intermetrics, Inc., Cambridge, Mass., March, 1970. N70-41238

[2]J. S. Miller, W. H. Vandever, S. F. Stanten, A. E. Avakian, and A. L. Koemala, "Final Report-- Engineering Study for the Functional Design of a Multiprocessor System", Intermetrics, Inc., Cambridge, Mass., September, 1972. N73-10235

[3]J. S. Miller, and W. H. Vandever, "Design Features of an Aerospace Multiprocessor", International Workshop on Computer Architecture, June 26-28, 1973. Grenoble, France.

2. MOTIVATION

2.1. PURPOSE: The system is oriented towards the general-purpose computational requirements of a manned, orbiting space station of about the 1980 time period. Its expected uses include real-time station control and data acquisition functions, plus interactive and batch data processing operations.

2.2. PHYSICAL ENVIRONMENT: The primary mission for which the system was designed is a spaceborne one. However, the design so far is limited to the functional specifications, and the physical environment considerations have had little impact on the configuration.

2.3. COMPUTING ENVIRONMENT: All connected elements will be aboard the space station. Components of the computer will be interconnected by dedicated buses. Terminals, displays, and sensors will be attached to the system by means of a multiplexed data bus.

2.4. COMPUTING OBJECTIVES: The performance requirements were rather soft. General objectives chosen for the system were real-time response of 5 milliseconds or better, and an equivalent processing rate of two million additions per second, for a three-processor configuration. Configuration flexibility was an important objective of the design.

2.5. RELIABILITY OBJECTIVES: Because no hardware was designed, no specific reliability requirements were imposed. The use of the system as the central computer for the space station life-support, trajectory, attitude, and experiment-processing functions places heavy emphasis upon a design which allows continued operation, even if at reduced performance, in the presence of faults. Although it is expected that brief outages of the system will be tolerable, our efforts have been directed at avoidance of all single-point failure modes.

2.6. DYNAMIC VARIABILITY: The multiplicity of processors is utilized to continue operation when processors fail. Failures of processors thus reduce the peak processing capacity. Similarly, a memory multiplexing scheme permits program and data mobility to work around loss of memory units. Increased multiplexing activity which follows removal of memory units from service also degrades maximum performance levels. Whether failures cause actual degradation in service depends upon the amount of excess capacity that was provided.

2.7. PENALTIES: Penalties from faulty operation are difficult to assess this early in the space station planning. However, loss of life is conceivably possible, but failure to achieve mission objectives is a more likely result of malperformance.

2.8. CONSTRAINTS: The constraints imposed by the space station environment will influence hardware design, but have not affected the functional design appreciably.

2.9. TRADEOFFS: The expected component and connection reliability will drive decisions relative to the level of excess processor, memory, and bus capability to be provided to achieve overall system availability goals.

3. DESCRIPTION

3.1. ARCHITECTURE

3.1.1. CONFIGURATIONS

3.1.1.1. INTERCONNECTIVITY: The basic configuration is shown in Figure 1. The internal configuration of a processor unit, showing duplicated elements and comparators, is given in Figure 2.

3.1.1.2. RANGE: Three to eight processors with at least as memory modules as processors, and preferably more to diminish conflict frequency. Given an environment where error detection was important but error recovery beyond instruction retry was less important, as few as one of each module can form a system.

3.1.1.3. CAPABILITY: The effective computing power of a one-processor system is about 0.6 Mips, or the approximate equivalent of a 360/65.

3.1.2. EXECUTIVE

3.1.2.1. MODES OF OPERATION: Software execution is based on a three-priority dispatching algorithm; of highest priority are the functions which require real-time response. These functions are kept short. Middle priority processes may be longer, but are interrupted only by real-time processes. Lowr batch-type processes are assigned lowest priority, and effectively run in a background mode.

3.1.2.2. SOFTWARE ORGANIZATION: The system software can run on any or multiple processors. Critical sections are protected by interlocks to avoid disruptions due to multiprocess interference. Executive and recovery software is stored in duplicate, under the hardware-implemented information protection scheme outlined below. Thus, single faults, even those which disable an entire memory module, cannot prevent access to or operation of the system.

3.2. FAULT TOLERANCE

3.2.1. FAULTS TOLERATED: Expressly attempted in the design is detection and recovery from every single fault. In this context, a second fault is one which occurs before the system recovery actions have been completed for the first fault. Processors, memories, and buses may be removed from operation as a result of faults. Performance capability is correspondingly reduced.

3.2.2. FAULTS NOT TOLERATED: Synchronized double faults in independent elements are not tolerated. Memory faults which affect information for which duplicate storage was judged unnecessary are not tolerated, although the size of this set of information is totally under user control. Flaws in system software may not be tolerated, but faults in applications software may adversely affect other processes only through disruption of data they share. The hardware supports detection of instruction loops, subscriptbounds violation, excessive processor time usage, and overtime inhibition of interruptions.

3.2.3. TECHNIQUES: Error detection and recovery are based on redundancy of information and capability. Processor units are comprised of duplicate elements, whose external signals are compared. This approach maximizes the error detection coverage, and minimizes the extra design needed for error-detection logic. Memories are parity checked for detection, and information whose loss cannot be tolerated is stored in duplicate. Processor local storage (M1) is all duplexed. Information in main memory (M2) is selectively duplexed by user specification. A novel use of descriptor-based storage management allows such duplication to be implemented entirely within the hardware. Software may be written without need for consideration of the detection or recovery problems. Duplicated buses support

comparison between independent copies for comprehensive error detection in the cases where duplicate storage is specified, and for error detection of transfers otherwise. The implementation of duplication in M2 causes copies to be kept in distinct units, so that even catastrophic failure of an entire unit can be tolerated. The memory multiplexing technique incorporated to reduce the total amount of M2 required for a given performance level allows program and data segments to be moved to new locations when M2 failures occur.

3.3. NOVELTY: With respect to fault-tolerance, a major attempt has been made to isolate application software from the effects of undetected hardware errors (by detecting as many as feasible), and from the necessity to devote explicit attention to survival following detected error (by providing adequate hardware support). With respect to architecture, a high-level instruction set has been developed, tailored to the needs of high-order language compilers, which are to be used to prepare all the software executed in the system. A novel approach to a descriptor-based, tagged-word design has been taken, in which the Multics paging strategy has been applied for the first time to variable-size pages (Burroughs' segments), a unified stack data format has been utilized, and tag-bits are incorporated into all necessary locations at a cost of at most one bit. Most words in the system need not expend bits on tags. Variable-length instructions are used to achieve maximum conciseness of program code.

3.4. INFLUENCES: The major influence on instruction-format and stack-organized processing came from the Burroughs B6700. The emphasis on hardware-implemented error detection and recovery resulted from adverse experience in attempting to provide these capabilities through software in the Apollo on-board guidance computer software development.

3.5. HARD-CORE: The only hard-core element in the system is the I/O controller. Because no degraded level of I/O capability seemed tolerable, the I/O controller is implemented with high internal redundancy, so as to be "failure proof".

4. JUSTIFICATION

4.1. RELIABILITY EVALUATION: Fault-tolerance is assessed by thought experiments, rather than simulations or other mechanical means. Reliability estimates cannot be made until hardware design commences.

4.2. COMPLETENESS OF EVALUATION: Evaluation consists of mental exercises. More rigorous exploration must necessarily await hardware design.

4.3. OVERHEAD: Because the hardware implements the bulk of the fault-tolerance provisions, very little price is paid for this capability in terms of performance. Segregation of processor local storage to M1 units to allow an alternate processor to rescue a failed one at any point in the execution of an instruction lengthens transit times somewhat. The hardware cost consists of a factor of two in processor costs, plus a bit for comparison circuits and error-control logic. Memory costs are increased by the amount that duplication of selected data requires extra storage capacity.

4.4. APPLICABILITY: The system described is applicable to any application where fault-tolerance is important. The emphasis on real-time capability makes it especially suitable for aircraft or process control applications.

4.5. EXTENDABILITY: Performance can be increased by use of faster components; memory sizes may be increased, etc. It is not believed that additional emphasis on fault-tolerance would be particularly productive.

4.6. CRITICALITIES: The absence of a rigid set of requirements has allowed a reasonable trade-off between conflicting factors. The design has been driven strongly only by the fault-tolerance requirements.

4.7. IMPLICATIONS: The system is designed around the concept that all software for the machine will be produced by correct compilers, which participate in the implementation and enforcement of operating system and programming ground rules. Use of high-level language for all software development is increasingly recognized as a valuable means of reducing software costs. However, the further advantages which can be achieved by intimate connections between compiler code-generation and system requirements have not been exploited.

5. CONCLUSIONS

5.1. STATUS: The functional design is complete.

5.2. EXPERIENCE: The design experience has been completely positive to date; the objectives and the approach continue to appear valid.

5.3. FUTURE: The project is now dormant due to NASA emphasis on the space shuttle program, which has caused space station planning to be heavily curtailed. Other sources for support of design continuation are being sought.

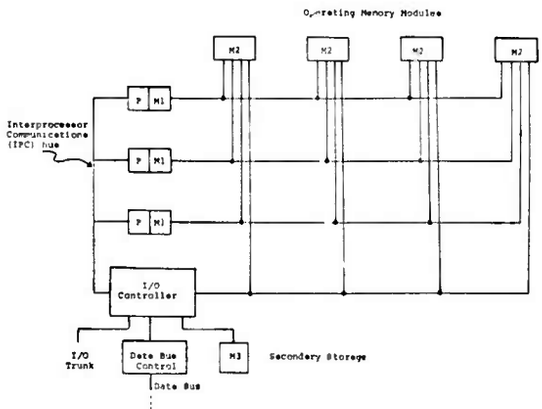


Figure 1: Basic Configuration: Three processors (P), and four operating memory modules (M2) are shown. M1 units contain local storage for each processor.

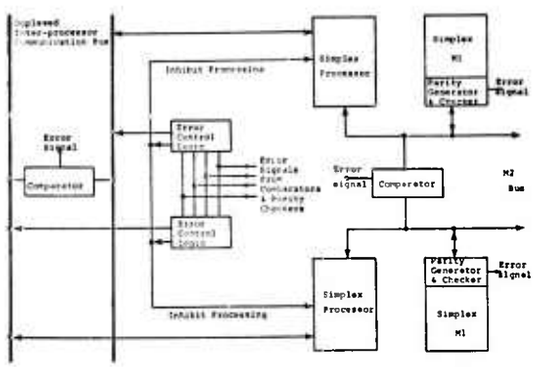


Figure 2: Internal Configuration of a Processor Unit (including M1 Local Storage)

SURVEY OF FAULT TOLERANT COMPUTING SYSTEMS

W. C. Ridgway III, Bell Labs, Madison NJ, July 1973

1. IDENTIFICATION
1.1 NAME: SAFEGUARD Data Processor

1.2 RESPONSIBILITY: Western Electric and BTL

1.3 SUPPORT: U. S. Army

1.4 PARTICIPANTS: Western Electric (Prime Contractor), Bell Laboratories (responsible for: system design, design of most digital equipment; and designed some system software), Univac (designed central processor and some diagnostic programs), IBM (designed some system software), Lockheed (designed system core memories).

1.5 START: Design effort for the ABM System was started in 1963.

1.6 COMPLETION: Hardware design is essentially complete; Software is in the final stages of development.

1.7 BIBLIOGRAPHY: (relevant to Fault Tolerance)
* D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," IEEE Transactions on Computers, Vol. C-21, No. 5, pp 464-71, May 1972.

* R. G. South, "A System for Simulating Faults in Large Logic Circuits," (Talk given at Lehigh University Workshop on Fault Detection and Diagnosis in Digital Systems, December 8, 1971).

* J. R. Hahn, "A Maintenance Approach for a Large Computer System," (Talk given at Lehigh University Workshop on Fault Detection and Diagnosis in Digital Systems, December 8, 1971).

2. MOTIVATION
2.1 PURPOSE: Part of Missile Defense System

2.2 PHYSICAL ENVIRONMENT: Ground Based

2.3 COMPUTING ENVIRONMENT: Interactive - real time - self-contained.

2.4 COMPUTING OBJECTIVES: To provide real-time detection, discrimination, tracking, and guidance functions required in a missle defense system.

2.5 RELIABILITY OBJECTIVES: To be able to withstand most system faults and still perform the defense mission.

2.6 DYNAMIC VARIABILITY: Design allows graceful degradation.

2.7 PENALTIES: Loss of defense capability.

2.8 CONSTRAINTS: Must operate in real-time in nuclear environment (e.g., high nuclear radiation levels and ground shock environment).

2.9 TRADEOFFS: Used (N + 1) redundancy and on-line automatic diagnostics instead of full equipment redundancy.

3. SYSTEM DESCRIPTION
3.1 ARCHITECTURE
3.1.1 CONFIGURATIONS
3.1.1.1 INTERCONNECTIVITY: See Figure 1.

3.1.1.2 RANGE: As noted in Figure 1
3.1.1.3 CAPABILITY: Classified

3.1.2 EXECUTIVE
3.1.2.1 MODES: Independent processors are not multi-programmable; however, the collective system is multiprogrammable and multiprocessing. There is no master-slave relationship between processors, therefore no hardcore (i.e., nonredundant critical hardware) exists. Programs are segmented into tasks which are dispatched by a scheduler.

Software Organization: I/O processing is performed asynchronously by a specific attached processor (known as I/O controllers). Executive, monitors, diagnostics, and other programs are run by the central processors as needed once prior tasks are completed.

3.2 FAULT TOLERANCE
3.2.1 FAULTS TOLERATED: The system is designed to withstand both transient and hard faults provided the problems in 3.2.2 are not met.

3.2.2 FAULTS NOT TOLERATED: The system can meet objectives unless either multiple faults occur simultaneously in enough different equipment, so that a viable system is not available, or transient faults (which are not isolated) affect critical units at an abnormally high rate.

3.2.3 TECHNIQUES: As shown in Figure 1, multiple units exist for each major type of equipment (e.g., memory). One of each type of these multiple units is included as a spare which may be substituted for any faulty unit. I.e., there are "n + 1" units of each type, where "n" is the number required to perform the tactical mission.

* System Reconfiguration of faulty units is controlled by special redundant status units described in Section 3.3.

* A special maintenance subsystem is employed which uses a separate maintenance path into all major registers (both data and control) in the system (see Figure 2). This subsystem is used to bootstrap the main system for normal initialization, to detect faults (via routine diagnostics), to isolate faults (using fault dictionaries), and to perform system recovery by detecting any catastrophic failure and reinitializing the system.

* Real-time diagnostics are periodically scheduled to detect equipment failures.

* Error detection and response features are designed into the normal system software. These features include defensive programming (e.g., data reasonableness checks), device managing (e.g., to isolate faulty units), initiating system rollback to a previously determined state, and calling for the maintenance subsystem to initiate complete system recovery (i.e., rollback to initial state)

* Redundant equipment is used (during routine surveillance periods) to play large scale system exercises against the on-line system. These exercises are valuable in uncovering errors, as well as maintaining the skill level of operators (and thus minimizing the possibility of manual errors).

* Parity is used to check data across interfaces and in memories.

* Error-detecting codes are used to check the transfer of critical data between some subsystems.

* The system master clock employs triple-modular redundancy to generate all major clock rates.

3.3 NOVELTY: The system has two significantly unique fault-tolerant design features as described below.

3.3.1 The first feature is the equipment status unit (SU) which controls the total system hardware configuration. The SU has the following characteristics and capabilities.

* There are two identical SU, either of which may be designated as the master unit.

* All communication paths between equipments (e.g., processor and memories) are controlled by the SU.

* By enabling (or disabling) the various communication paths, the SU can split the system into two separate computers (e.g., one can be used to exercise the other). Individual equipment can also be isolated, if necessary, to allow diagnostics to be performed.

* The error detection circuits in each equipment send reports to the SU. These reports are used by software to determine what equipment should have diagnostics performed, as well as to make reconfiguration determinations.

* The SU enables special test paths in each equipment so that diagnostics may be performed as described in 3.3.2.

3.3.2 The second unique fault-tolerant design feature centers around the maintenance subsystem. The characteristics and capabilities of this subsystem are:

SURVEY OF FAULT TOLERANT COMPUTING SYSTEMS

W. C. Ridgway III, Bell Labs, Madison NJ, July 1973

1. IDENTIFICATION
1.1 NAME: SAFEGUARD Data Processor

1.2 RESPONSIBILITY: Western Electric and BTL

1.3 SUPPORT: U. S. Army

1.4 PARTICIPANTS: Western Electric (Prime Contractor), Bell Laboratories (responsible for: system design, design of most digital equipment; and designed some system software), Univac (designed centeral processor and some diagnostic programs), IBM (designed some system software), Lockheed (designed system core memories).

1.5 START: Design effort for the ABM System was started in 1963.

1.6 COMPLETION: Hardware design is essentially complete; Software is in the final stages of development.

1.7 BIBLIOGRAPHY: (relevant to Fault Tolerance)
* D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," IEEE Transactions on Computers, Vol. C-21, No. 5, pp 464-71, May 1972.

* R. G. South, "A System for Simulating Faults in Large Logic Circuits," (Talk given at Lehigh University Workshop on Fault Detection and Diagnosis in Digital Systems, December 8, 1971).

* J. R. Hahn, "A Maintenance Approach for a Large Computer System," (Talk given at Lehigh University Workshop on Fault Detection and Diagnosis in Digital Systems, December 8, 1971).

2. MOTIVATION
2.1 PURPOSE: Part of Missile Defense System

2.2 PHYSICAL ENVIRONMENT: Ground Based

2.3 COMPUTING ENVIRONMENT: Interactive - real time - self-contained.

2.4 COMPUTING OBJECTIVES: To provide real-time detection, discrimination, tracking, and guidance functions required in a missle defense system.

2.5 RELIABILITY OBJECTIVES: To be able to withstand most system faults and still perform the defense mission.

2.6 DYNAMIC VARIABILITY: Design allows graceful degradation.

2.7 PENALTIES: Loss of defense capability.

2.8 CONSTRAINTS: Must operate in real-time in nuclear environment (e.g., high nuclear radiation levels and ground shock environment).

2.9 TRADEOFFS: Used (N + 1) redundancy and on-line automatic diagnostics instead of full equipment redundancy.

3. SYSTEM DESCRIPTION
3.1 ARCHITECTURE
3.1.1 CONFIGURATIONS
3.1.1.1 INTERCONNECTIVITY: See Figure 1.

3.1.1.2 RANGE: As noted in Figure 1
3.1.1.3 CAPABILITY: Classified

3.1.2 EXECUTIVE
3.1.2.1 MODES: Independent processors are not multi-programmable; however, the collective system is multiprogrammable and multiprocessing. There is no master-slave relationship between processors, therefore no hardcore (i.e., nonredundant critical hardware) exists. Programs are segmented into tasks which are dispatched by a scheduler.

Software Organization: I/O processing is performed asynchronously by a specific attached processor (known as I/O controllers). Executive, monitors, diagnostics, and other programs are run by the central processors as needed once prior tasks are completed.

3.2 FAULT TOLERANCE
3.2.1 FAULTS TOLERATED: The system is designed to withstand both transient and hard faults provided the problems in 3.2.2 are not met.

3.2.2 FAULTS NOT TOLERATED: The system can meet objectives unless either multiple faults occur simultaneously in enough different equipment, so that a viable system is not available, or transient faults (which are not isolated) affect critical units at an abnormally high rate.

3.2.3 TECHNIQUES: As shown in Figure 1, multiple units exist for each major type of equipment (e.g., memory). One of each type of these multiple units is included as a spare which may be substituted for any faulty unit. I.e., there are "n + 1" units of each type, where "n" is the number required to perform the tactical mission.

* System Reconfiguration of faulty units is controlled by special redundant status units described in Section 3.3.

* A special maintenance subsystem is employed which uses a separate maintenance path into all major registers (both data and control) in the system (see Figure 2). This subsystem is used to bootstrap the main system for normal initialization, to detect faults (via routine diagnostics), to isolate faults (using fault dictionaries), and to perform system recovery by detecting any catastrophic failure and reinitializing the system.

* Real-time diagnostics are periodically scheduled to detect equipment failures.

* Error detection and response features are designed into the normal system software. These features include defensive programming (e.g., data reasonableness checks), device managing (e.g., to isolate faulty units), initiating system rollback to a previously determined state, and calling for the maintenance subsystem to initiate complete system recovery (i.e., rollback to initial state)

* Redundant equipment is used (during routine surveillance periods) to play large scale system exercises against the on-line system. These exercises are valuable in uncovering errors, as well as maintaining the skill level of operators (and thus minimizing the possibility of manual errors).

* Parity is used to check data across interfaces and in memories.

* Error-detecting codes are used to check the transfer of critical data between some subsystems.

* The system master clock employs triple-modular redundancy to generate all major clock rates.

3.3 NOVELTY: The system has two significantly unique fault-tolerant design features as described below.

3.3.1 The first feature is the equipment status unit (SU) which controls the total system hardware configuration. The SU has the following characteristics and capabilities.

* There are two identical SU, either of which may be designated as the master unit.

* All communication paths between equipments (e.g., processor and memories) are controlled by the SU.

* By enabling (or disabling) the various communication paths, the SU can split the system into two separate computers (e.g., one can be used to exercise the other). Individual equipment can also be isolated, if necessary, to allow diagnostics to be performed.

* The error detection circuits in each equipment send reports to the SU. These reports are used by software to determine what equipment should have diagnostics performed, as well as to make reconfiguration determinations.

* The SU enables special test paths in each equipment so that diagnostics may be performed as described in 3.3.2.

3.3.2 The second unique fault-tolerant design feature centers around the maintenance subsystem. The characteristics and capabilities of this subsystem are:

* It is a special subsystem consisting of its own small computer and associated equipment whose sole purpose is to perform fault detection, isolation, and recovery. This is done in conjunction with the status unit described in section 3.3.1.

* A special maintenance path is designed into all digital equipment. The sole function of this path is to allow diagnostics; this path, in essence, allows breaking large sequential blocks of logic into smaller combinational blocks which are easier to diagnose for faults.

* The maintenance system uses fault dictionaries to isolate faults. The dictionaries are searched by using error patterns which are collected from tests on the hardware. The dictionaries are created by a table-driven logic-path sensitized, logic simulator, which predicts possible error responses when the diagnostics are run against the simulated hardware.

* The maintenance subsystem is redundant; each half performs diagnostics on the other half.

3.4 INFLUENCE: Main system concepts and architecture were developed by numerous members of staff within Bell Labs.

3.5 HARD-CORE: There is no hard-core in the sense of non-redundant hardware.

4. SYSTEM JUSTIFICATION

4.1 RELIABILITY EVALUATION: System "reliability" is expressed as A*R (i.e., availability-reliability" product), where A*R is defined as "the probability that the system

* will be available (i.e., error free) for full-load operations when needed, and

* will continue to operate without system failure throughout the engagement."

The total system A*R requirement is budgeted across the various subsystems. Analysis of hardware reliability models for each subsystem indicates that the design should meet the A*R requirements. Diagnostic programs are analyzed by the logic simulator (which also creates the fault dictionaries) to determine the percent of faults detected. This information is used in the A*R models. In addition, error control software as well as diagnostic programs are further tested by inducing randomly selected faults in the hardware.

4.2 COMPLETENESS OF EVALUATION: Analysis of the system design by the hardware A*R models has been completed, except for possible updating based on diagnostic programs analysis results. About 20% of the diagnostic programs have been analyzed by the logic simulator. About 15% of the fault insertion analysis has been completed.

4.3 OVERHEAD: For the largest system, about 10% of the hardware is required for the maintenance subsystem. Other error detection hardware and equipment redundancy account for possibly another 15% of the hardware. Diagnostic programs and defensive programming techniques are estimated to account for about 25% of the total source instructions.

4.4 APPLICABILITY: One or more of the fault tolerant system features may be used with most digital systems.

4.5 EXTENDABILITY: It would probably be impractical to extend any of the fault tolerant concepts in the present system.

4.6 CRITICALITIES: System cost could be significantly reduced by a relaxation in reliability. Multiprogramming and multiprocessing are both required to handle the required processing tasks in real time.

4.7 IMPLICATIONS: Because of the stringent availability requirements, automatic fault detection, isolation and recovery were of primary consideration.

5. CONCLUSIONS

5.1 STATUS: The prototype system is now being integrated.

5.2 EXPERIENCE: Experience to date indicates that the system should meet its availability/ reliability goals.

5.3 FUTURE: The system is scheduled for installation at various government facilities in the near future.

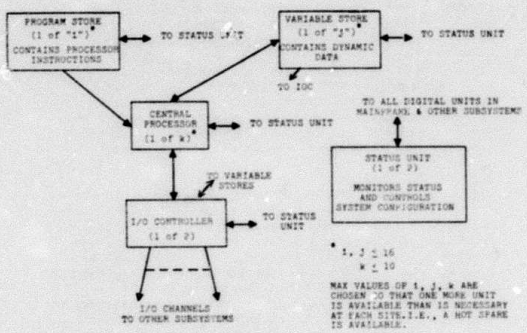


FIGURE 1 - COMPUTER MAINFRAME

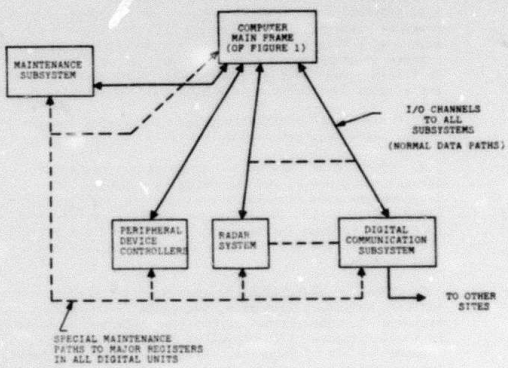


FIGURE 2 - OVERALL DIGITAL SYSTEM STRUCTURE

SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS

Prof. Jerome H. Saltzer
Project MAC, MIT, Cambridge, MA., April 1973

1. IDENTIFICATION
1.1. NAME: Multics, for MULTiplexed Information and Computing Service.

1.2. RESPONSIBILITY: Massachusetts Institute of Technology, Project MAC, Computer Systems Research Division. As of 1/73, a Honeywell product.

1.3. SUPPORT: Advanced Research Projects Agency via Office of Naval Research.

1.4. PARTICIPANTS: MIT Project MAC; Honeywell Cambridge Information Systems Laboratory (formerly the General Electric Company Computer Department). Also Bell Telephone Laboratories, 1965-69.

1.5. START: Planning in 1963-4, complete proposal in 1965.

1.6. COMPLETION: System first usable in 1968, available for public use at M.I.T. in 1969, now commercially available. Research continuing.

1.7. BIBLIOGRAPHY:
*The Multiplexed Information and Computing Service: Programmers' Manual, M.I.T. Project MAC, Rev. 13, 1973.

*F.J. Corbato, et al., Session 6: A new remote accessed man-Machine system, AFIPS Conf. Proc. 27, (FJCC 1965), Spartan Books, Washington D.C., pp. 185-247.

*F.J. Corbato, J.H. Saltzer, and C.T. Clingen, "Multics -- the first seven years," AFIPS Conf. Proc. 40, (SJCC 1972), AFIPS Press, Montvale, N.J., pp. 571-583.

*E.I. Organick, The Multics System: an Examination of its Structure, MIT Press, 1972.

2. MOTIVATION
2.1. PURPOSE: Multics is a prototype of the general-purpose computer utility. It is intended to allow interactive access to a shared information base, permit use of general purpose programming, and be expandable and evolvable. Reliability and fault tolerance were considered to be only two of many overlapping and conflicting objectives.

2.2. PHYSICAL ENVIRONMENT: Multics is designed for use in a ground-based data processing center.

2.3. COMPUTING ENVIRONMENT: Multics is approached by interactive displays and typewriter terminals. For large-volume data processing applications, card, printer, and magnetic tape peripherals are provided, but all job initiation is done interactively. Terminals are attached directly, via the dial-up telephone network, and via the ARPA network.

2.4. COMPUTING OBJECTIVES: Multics provides a wide range of software services, languages, and tools for constructing programs and subsystems. It provides interactive response to small requests at the level of 2 seconds average, 5 seconds for 90% of requests. Larger compute-bound requests are scheduled at a lower priority. With initial (1964) hardware, configurations supporting from 10 to 120 simultaneous users can be constructed. Hardware installed in fall 1972 increases the potential limit to about 400 users, and also improves response time. Software design range is from 10 to 1000 users.

2.5. RELIABILITY OBJECTIVES: The primary reliability objective concerns integrity of on-line file storage. Ideally, the user can rely on the system to have a perfect memory for his files. A secondary availability objective is that the system operate continuously, on a 24-hour per day basis. Recovery time following a failure is permitted to have a wide variation, but an average on the order of a few minutes. Objectives such as 100% continued operation in the face of any single failure were not attempted.

2.6. DYNAMIC VARIABILITY: An individual installation may choose the fraction of system resources to be used for file backup operation, thereby providing varying degrees of maximum setback for its users following the worst possible kind of a system crash. If 20% of resources are used for backup, a maximum of 30 minutes of work can be lost by a system crash. Smaller quantities of backup can produce larger setbacks. The design is multiprocessor, to permit restart with a smaller, lower-performance configuration, without waiting for hardware to be repaired.

2.7. PENALTIES: Penalty depends on the range of applications for which the system is being used. In the M.I.T. environment, loss of stored files or lack of system availability may mean disruption of administrative and departmental operations which use the system. For programming use, the penalty is small.

2.8. CONSTRAINTS: Multics is intended to be economically competitive with other commercial and scientific data processing systems. No unusual physical constraints exist.

3. DESCRIPTION
3.1. ARCHITECTURE
3.1.1. CONFIGURATIONS
3.1.1.1. INTERCONNECTIVITY: The hardware (Honeywell 600/6000) is a multiprocessor, multimemory design in which each processor is connected by a separate cable to each memory box. I/O controllers are attached to the memory boxes in the same way as processors.

3.1.1.2. RANGE: Software permits 1-10 processors, 128K to 16M 36-bit words without change. Small changes would permit essentially unlimited (e.g., 10E14 words) memory sizes. Current hardware permits 1-7 processors, 128K to 2M 36-bit words. Small changes in hardware would permit up to 16M words.

3.1.1.3. CAPABILITY: Honeywell 645 CPU runs at 330,000 instructions/sec, about half the speed of a 360/65. Honeywell 6180 CPU is expected to run about 1M instructions/sec, somewhere between a 370/155 and a 370/165.

3.1.2. EXECUTIVE
3.1.2.1. MODES: System permits user-constructed cooperative processes, utilizing multiprocessors and multiprogramming. The multiple processors run independently and autonomously rather than in a master/slave processor organization.

3.1.2.2. SOFTWARE: The system software appears to each user as a private supervisor residing within his personal address space. A small section of the supervisor is core resident; the rest of the supervisor as well as all user programs and data are paged. All programs, including the core resident supervisor, are in the virtual memory.

3.2. FAULT TOLERANCE
3.2.1. FAULTS TOLERATED: All forms of hardware and software failures which are severe enough to cause a system crash result in a service outage ranging from a few minutes to a few hours, followed by availability of a reinitialized system. All files are preserved, but computations in progress must be restarted from the beginning or from the last checkpoint which the user has provided. If the operations staff has been well-organized in protecting tape copies, it is possible to completely and automatically recover even from a fire which destroys the computer system (given enough time to install replacement hardware).

3.2.2. FAULTS NOT TOLERATED: Failures involving physical destruction of on-line storage devices (e.g., disk head crashes) are tolerated, but can result in outage of up to several hours while reconstruction of the on-line files from back up copies is performed.

3.2.3. TECHNIQUES: Backup copying: When an on-line file is created, within a half an hour, a backup copy is automatically made on a journal tape. Once each day, an extra set of journal tapes are independently written, containing copies of all files created since the previous day. Once each week, a logical copy of every on-line file is made onto tape, to limit the number of journal tapes which must be scanned to reconstruct the on-line files. (The times of 1/2 hour, 1 day and 1 week are adjustable by the installation to local needs.)

*Salvaging: Following a system crash for any reason, a salvager program inspects the condition of all on-line files and directories, and reports any uncorrectable inconsistencies or irregularities in content and format. A small amount of redundancy is used in directory structures, to assist the salvager.
*On-line salvaging: Whenever an inconsistent directory entry is discovered during normal operation, a version of the salvager is immediately invoked to correct the situation. Normally, service is not interrupted.
*Retrieval: If the salvager finds it impossible to reconstruct one or a few files, but the number is small enough that the expense of a complete file system reconstruction from backup tapes is not warranted, the user of the file is notified, and he may initiate retrieval of his file from the backup or journal tapes. Retrieval of older copies may also be requested by the user if he accidentally damages or deletes the current on-line copy of a file.
*Continuous Operation: The system is dynamically reconfigurable, which means that processors and memory boxes may be added or removed while the system is running a production load. This technique permits both hardware and software maintenance to be performed on detached units. Since in addition the software system may be loaded onto any available configuration of processors and memory boxes, recovery following a solid hardware failure can be very rapid.

3.3. NOVELTY: The primary novelty of Multics in this area is that the reliability objectives have been integrated into a general-purpose computer programming system which also meets a wide variety of other objectives. As far as is known, Multics is the first general purpose system to permit dynamic reconfiguration of processors and memory.

3.4. INFLUENCES: Experience in designing and using the Compatible Time Sharing System for the IBM 7094 provided the most obvious influence. The multiprocessor organization was influenced by the Burroughs DB25 computer system.

4. JUSTIFICATION
4.1. RELIABILITY EVALUATION: In an operational environment at M.I.T. for several years, the rate of loss of files because of system failures has been low enough to be acceptable to the user community, but has not been evaluated. The average time down when a failure occurs is about 10-15 minutes.

4.2. OVERHEAD: Hardware negibly redundant. Variable software overhead for backup. (See 2.6.)

4.7. IMPLICATIONS: For the file backup procedure to be effective, it is essential that the computer operating staff be highly organized, and that the operations management thoroughly understand its responsibility in helping safeguard user files stored on-line. (For example, sloppy tape storage management cannot be tolerated.)

5. CONCLUSIONS
5.1. STATUS: The system has been operational at M.I.T. for 4 years and is the primary time-sharing system there. It is also in use at 3 other sites, on order at several others.

5.2. EXPERIENCE: The design seems to be adequate for the quantity of storage currently being managed (100 million words), but maximum reload times are proportional to this quantity of on-line storage and are near the limit of tolerance. A revised reload strategy employing parallel processes is expected to provide an order of magnitude increase in the practical storage quantity limit.

5.3. FUTURE: Research on many aspects of computer operating systems other than reliability is continuing, using Multics as a laboratory vehicle.

SURVEY OF FAULT TOLERANT COMPUTING SYSTEMS

Werner Ulrich, Bell Labs, Inc.
Naperville, Illinois 65540 June 1972

1. IDENTIFICATION
1.1. NAME: No. 1 ESS. A number of electronic switching systems have been designed by Bell Laboratories during the past several years. Those which have been described in the literature include No. 1 ESS, No. 101 ESS, No. 2 ESS and the Traffic Service Position Systems (TSPS). This response will be concerned exclusively with No. 1 ESS, a large telephone central office designed primarily for service application.

1.2. RESPONSIBILITY: The Indian Hill Switching Division, Naperville Laboratory of Bell Laboratories.

1.3. SUPPORT: Development of the System supported by Western Electric Company (WE), the manufacturing unit of the Bell System.

1.4. PARTICIPANTS: The system was designed and developed by Bell Laboratories, is manufactured and installed by the Western Electric Company and is operated by the various Bell System operating companies.

1.5. START: Active work on the design of No. 1 ESS began in late 1959.

1.6. COMPLETION: The first system was put into service in Succasunna in 1964. Both hardware and software improvements have been made in the system from that time.

1.7. BIBLIOGRAPHY: The basic description of No. 1 ESS is in the September, 1964 issue of the Bell System Technical Journal. In addition the following bibliography deals specifically with the problems covered in this survey.

* Downing, R. W., et al., "No 1 ESS Maintenance Plan," Bell System Technical Journal, Vol. 43, no. 1961-2020, September, 1964.

* Beuscher, H. J., et al., "Administration and Maintenance Plan of No. 2 ESS," Bell System Technical Journal, Vol. 48, pp. 2765-2825, October, 1969.

* Chang, H. Y. and Thomas W., "Methods of Interpreting Diagnostic Data for Locating Faults in Digital Machines," Bell System Technical Journal, Vol. 46, pp. 289-318, February, 1967.

* Toiang, S. H., Maugk, G. and Seckler, H. N., "Maintenance of a Large Electronic Switching System," IEEE Transactions on Communications Technology, pp. 1-9, February, 1969.

* Aitcheson, E. J. and Cook, R. F., "No. 1 ESS ADF Maintenance Plan," Bell System Technical Journal, Vol. 49, No. 10, pp. 2831-2856, December, 1970.

* Nowak, J. S. and Tuomenokas, L. S., "Memory Mutilation in Stored Program Controlled Telephone System," 1970 IEEE International Conference of Communications, pp. 43-32-43-45.

* Chang, H. Y. and Scanlon, J. M., "Design Principles for Processor Maintainability in Real-Time Systems," Proceedings of Fall Joint Computer Conference, pp. 319-328, 1969.

* Nowak, J. S., "Emergency Action for No. 1 ESS," Bell Laboratories Record, Vol. 49, No. 6, pp. 176-179, June/July, 1971.

* Connet, J. R., Pasternak, E. J. and Wagner, R. D., "Software Defenses in Real-Time Control Systems," Second Annual International Symposium on Fault Tolerant Computing, June 19-21, 1972, Boston, Massachusetts.

* Alrquist, R. T., et al, "Software Protection in No. 1 ESS," 1972 IEEE Conference on Communications, June, 1972.

* Ketchledge, R. W., "Service Experience with No. 1 ESS Equipment," International Conference on Electronic Switching, 1966 Proceedings, Paris, Edition Chiron, pp. 712-716.

* Vaughan, H. E., "Experience with the No. 1 ESS," International Conference on Electronic Switching, 1966 Proceedings, Paris, Edition Chiron, pp. 704-711.

* Hauck, G., "Early No. 1 ESS Field Experiences, Part 1, 2-Wire System for Commercial Implications," IEEE Transactions on Communications Technology, Vol. 15, pp. 744-750, December, 1967.

* Seckler, H. N., "Early No. 1 ESS Field Experience, Part 2, 4-Wire System for Government and Military Implications," IEEE Transactions on Communications Technology, Vol. 15, pp. 751-754, December, 1967.

* Johannesen, J. D., "No. 1 ESS Service Experience - Software," IEEE Conference on Switching Techniques for Telecommunication Networks, Conference Publication No. 52, pp. 459-462, April, 1969.

* Stechler, R. E., "No. 1 ESS Service Experience - Hardware," IEEE Conference on Switching Techniques for Telecommunication Networks, Conference Publication No. 52, pp. 463-466, April, 1969.

2. MOTIVATION

2.1. PURPOSE: Control the setting up and disconnection of calls between telephone customers attached to the system or between these telephone customers and other customers in distant central offices.

2.2. ENVIRONMENT: The system must operate in the presently existing telephone plant and must communicate with telephone customers and other existing central offices.

2.3. COMPUTING ENVIRONMENT: The system does internal data processing relating the signals transmitted by customers and by other central offices to the desired telephone connections. Its inputs are these signals as gathered by peripheral equipment associated with the central processing unit and its outputs are control signals to a telephone switching network and output signals which are transmitted to distant central offices.

2.4. COMPUTING OBJECTIVES: The basic objective of the system was to handle 100,000 peak busy hour calls. While the original version of the system did not meet this goal, software improvements have allowed this goal to be met during the past year.

2.5. RELIABILITY OBJECTIVES: Reliability objective for the system was a down time of no more than 2 hours in 40 years. When the down time objectives were originally set, this down time was predicted to be due primarily to simultaneous hardware failures of duplicated processor units. As it turned out, software failures or human failures leading to massive memory mutilation have been the primary source of down time. In recent years, the down time has been approaching the range of 10-15 hours per 40 years and is still going down from this point.

2.6. DYNAMIC VARIABILITY: Reliability in 2.5 above has been defined in terms of total system reliability. Dynamic variability can be thought of in terms of the ability to handle telephone traffic in the presence of overload exceeding the capability of the system. A dynamic overload response has been built into the system which allows additional service requests to be throttled during periods of excessive demand.

2.7. PENALTIES: Penalties for total system failure may include the inability to make a telephone call at a critical time, with resultant possible loss of life and/or property. For example, the inability to call the fire department can be quite serious. However, the penalty is dependent on the time of the occurrence of the failure. In many cases no penalty will result.

2.8. OFFICE CONSTRAINTS: The equipment must be installed in a telephone central office. It is desirable that it operate with normal Bell System nominal 48-volt battery as the primary power source. Minimum space is desirable but not critical since the cost of space is comparable to the normal cost of office and factory space. Air conditioning is normally provided but the system must be able to work for moderate periods of time without air conditioning. The cooling system consists of normal convection cooling augmented by conventional air conditioning.

2.9. TRADEOFFS: System capability and system storage costs are among the main tradeoffs available in the system. The user of a read-only program memory means that all program storage must be paid for on a permanent basis. The range of office sizes encountered in the Bell System means that a change in system capacity will affect the market for a No. 1 ESS. Price was a very important factor since a No. 1 ESS provides the same basic type of telephone service available from older, efficient, and relatively inexpensive telephone systems. Price differential must be justified in terms of greater flexibility for future changes and long term lower costs due to automated manufacturing techniques.

3. DESCRIPTION OF THE SYSTEM

3.1. ARCHITECTURE

3.1.1. CONFIGURATIONS

3.1.1.1. INTERCONNECTIVITY: The basic block diagram of the system is presented in the BSTJ reference (first article). Basically each central control has three bus systems: a peripheral bus system, including an addressing system, a unit selection system, and a response bus; a read-write store (call store) bus system with addressing, data-write, and data-read sections; and a read-only memory (program store) bus system including addressing and response information. Each of the central controls has full access to all busses. The two central controls are interconnected by match busses to allow information in the two controls to be matched. In the normal mode only one central control has control access to the peripheral bus system although both central controls listen to the response bus. Each of the central controls in the normal mode controls one set of a duplicate set of stores. However, it is possible for one central control to control all stores and for the central controls to alternate in controlling the peripheral bus system. All critical equipment which includes stores, central controls, busses and peripheral control units are duplicated.

For larger systems, a signal processor is placed on the call store bus. This signal processor has access to its own read-write memories and also has access to the peripheral bus system. The signal processor then is used to control input/output equipment such as signaling equipment and the switching network.

3.1.1.2. RANGE: Only one basic central processor is used in any system, defining a central processor as a duplicated central control, duplicated signal processors when required, and duplicated stores. The duplication of the memory modules is such that each module is effectively divided into two parts; therefore, an odd number of modules can exist in the system. The limit on the number of read-only stores including duplication is 12, each of which contains 131,000 44-bit words (the 44 bits are 37 bits of information and 7 bits of Hamming code); the limit on the number of call store modules is about ten, each containing 32,000, 24 bits per word. The original system contained 8,000 word modules, but this year we have started using the larger sizes.

3.1.1.3. CAPABILITY: The best way of indicating the capacity of processors is in terms of the number of calls which can be handled; as indicated earlier this figure now exceeds 100,000 during the peak busy hour. The basic cycle time of the system is 5.5 microseconds during which a complete addition can be performed. Program and data can be read in parallel. The order structure of the system is sufficiently powerful that the 5.5 microseconds time give a misleading by low indication of the basic power of the processor. In general terms, it might be compared in power to an IBM 7094 computer.

3.1.2. EXECUTIVE AND OPERATING SYSTEMS

3.1.2.1. MODES OF OPERATION: The signal processor operates independently of the central control. The central control handles all telephone calls in the office on a time shared basis working on one call at a time but only doing part of the work necessary to process that call. Work is time sliced so that, in general, no single task should exceed about 20 milliseconds of processor time. In an office without a signal processor, I/O is carried out by an interrupt level program which takes command of the system every milliseconds.

3.1.2.2. SOFTWARE ORGANIZATION: Tasks are dispensed by an executive program which checks individual task request hoppers to see if the I/O programs have discovered work of a particular category. Task hoppers are examined at different frequencies. An overall check is made to insure that all task hoppers are examined regularly. The main interactions between the executive and specific hardware are: interrupts which are used for I/O operations and trouble detection and analysis; and the emergency action circuit which recognizes failure of the system to cycle satisfactorily and automatically causes a switch to standby equipment.

3.2. FAULT TOLERANCE

3.2.1. FAULTS TOLERATED: The system can tolerate faults in duplicated equipment by simply switching to the appropriate standby equipment. Because of the full duplication, this does not affect system performance.

3.2.2. FAULTS NOT TOLERATED: The system is moderately sensitive to marginal conditions and intermittent faults. If these conditions occur sufficiently frequently that they interfere with system performance but not so often that they are caught by fault check programs, system performance may suffer. Because of the large circuit margins which were used in the design of the system, this type of situation has not occurred too frequently. (Note the very low down time discussed above.)

The system is, of course, sensitive to faulty software. Such software can, under the right circumstances, cause massive memory mutilation which then requires reinitialization. Reinitialization is not required of telephone calls already up in the system whose associated memory records are still consistent. It must be understood that all the software in the system is stored in read-only memory and is checked thoroughly before being placed into service. Faulty software, therefore, is software whose faults were not discovered during a very complete system checkout and which occur only under unusual circumstances, frequently with the help of a hardware error. Normal program errors are caught before they are installed in the field. No programmer has direct access to programs actually in the field.

3.2.3. TECHNIQUES

* Duplication. All critical equipment is duplicated.

* Matching. The information in duplicated equipment is matched. This offers almost immediate trouble detection.

* Information stored in the read-only memory is stored in single error correcting double error detecting Hamming code which covers data and address information. This means that a word read from an address different from the address central control is detectable.

* Fault Check Routine. As soon as a trouble is detected a program controlled examination is made to see which of two duplicate units has a potential trouble.

* Diagnostics. Diagnostic programs run under the control of a working processor configuration on a unit which has been switched out by fault check routines or whose diagnostic has been requested by maintenance personnel.

* Audits. Checks are periodically made on most of the information recorded in the read/write store to look for consistency. Audits include checks of pointers in the system, and checks of the completeness and consistency of linked lists.

* Software Emergency Action. An overall software check is made to make sure that the system is cycling through all tasks properly and at a normal interval. If the tests fail, progressively stronger initialization of the system is used.

* Hardware Emergency Action. If the system gets into a state such that the software emergency action programs cannot cycle properly, hardware emergency action switches to standby equipment and tries to find a working configuration consisting of a sound program store, call store, and central control for controlling subsequent system maintenance programs.

3.3. NOVELTY: I believe the novelty of the system lies not so much in any of the individual items mentioned above but in the extremely rigorous effort made throughout the system to get a very high degree of reliability. An attempt was made not only to detect all troubles and automatically switch to standby equipment, but to automatically diagnose all troubles. The latter function was less successful than the former although a surprisingly large portion of the system troubles can be accurately diagnosed automatically.

3.4. INFLUENCES: Current switching systems have been influenced by and have influenced the development of computer systems.

3.5. HARD CORE: I define hard core as that equipment which if it fails will make it impossible for the system to continue operation. We have tried very hard to minimize such equipment and the fact that we have not had a complete system outage to the presence of a single hard fault in such equipment implies that the amount of hard core equipment is relatively small.

Hard core equipment is sometimes defined in terms of diagnostic ability. There is no question that we cannot diagnose all solid troubles automatically.

4. JUSTIFICATION

4.1. RELIABILITY EVALUATION: The basic reliability evaluation of the system is made by examining the performance of the systems in the field. We have now accumulated in excess of a million hours of system operation. Records of system outages are kept automatically because these outages are printed on the teletypewriter output of the system.

4.2. COMPLETENESS OF EVALUATION: See above.

4.3. OVERHEAD: Considering the fact that much of the system is not duplicated (for example, the switching network is not duplicated); that the cost of duplication is greater than the simple doubling of the cost of all basic equipment since the facilities for duplication and for switching must also be included; it might be fair to state in overall terms that about 50 percent of the cost of the system is in some way due to the requirement that we must provide the ability to continue operation in the presence of trouble.

4.4. APPLICABILITY: The system is designed for telephone applications. The basic principles can be used in any application in which high reliability is important.

4.5. EXTENDABILITY: See above.

4.6. CRITICALITIES: The system was custom designed to a very high degree to meet the basic objectives set at the time system design was started, and taking into account the technology of that time.

4.7. IMPLICATIONS: A high degree of custom design places substantial restrictions on the application programmers if they are to use the system effectively.

5. CONCLUSION

5.1. STATUS: System is operational and is gaining wide acceptance in the Bell System.

5.2. EXPERIENCE: Achievement and maintenance of high reliability system is a continuous process requiring considerable and continuing effort especially if the applications of the system continue to change and expand.

SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS

Prof. Daniel Siewiorek, Computer Science Dept, Carnegie-Mellon University, Pittsburgh, Pa. 15213, April 1973

1. IDENTIFICATION
1.1. NAME: C.mmp (multi-mini-processor)

1.2. RESPONSIBILITY: Computer Science Department, Carnegie-Mellon University

1.3. SUPPORT: ARPA
1.4. PARTICIPANTS: C.G. Bell, B. Broadley, E. Cohen, A. Jones, R. Levin, J. McCredie, A. Newell, C. Pierson, F. Pollack, R. Reddy, W. Wulf, and many others.

1.5. START: Mid-1971
1.6. COMPLETION: Mid-1973 (hardware, initial software)
1.7. BIBLIOGRAPHY:
* Bell, C.G., W. Broadley, W. Wulf, A. Newell, C. Pierson, R. Reddy, and S. Rege, "C.mmp: The CMU Multi-mini-processor Computer - Requirements and Overview of the Initial Design," August, 1971. Carnegie Mellon University, Computer Science Department Research Report. (AD 739963)
* W. Wulf, "C.mmp: A Multiminiprocessor," Computer Science Research Review, Carnegie-Mellon University, 1971-1972.
* S. Fuller, R. Swan, W. Wulf, The Instrumentation of C.mmp, A Multiminiprocessor, COMPCON 73, pp. 173-176, 1973.

2. MOTIVATION
2.1. PURPOSE: General-purpose and real-time computing
2.2. PHYSICAL ENVIRONMENT: Ground-based
2.3. COMPUTING ENVIRONMENT: Initially stand-alone; eventually on the ARPANET.

2.4. COMPUTING OBJECTIVES: C.mmp was designed to provide a real-time processing and time sharing environment, e.g., for research in speech and vision. Thus special high data rate, real-time interfaces are required to acquire speech and vision data from the external environment. Also, real-time processing for the speech-understanding system is an ultimate goal. Execution of up to 3 to 15 million instructions/sec achieved through 1-16 memory modules (650 nsec cycle time) with up to 256K words each, 1-16 processors (PDP-11), 16x16 crossbar switch with 80x10E6 words/second. capacity.

2.5. RELIABILITY OBJECTIVES: Since the system is ground based and maintenance is available, the major reliability objective is high availability. With the ability to dynamically reconfigure the system, the ultimate goal is continuous availability.

2.6. DYNAMIC VARIABILITY: Reliability can be traded for performance by 1) parallel and independent computations on different processors and/or by 2) graceful degradation, possibly even on a millisecond scale.

2.7. PENALTIES: Mutilation of data in critical system tables could cause a system crash. Loss of experimental data or active programs would result.

2.8. CONSTRAINTS: The major constraint was cost. The objective was to build a high-performance system using off-the shelf components which could out-perform conventional systems for a fraction of the cost. The presence of multiple copies of various components in the system also provides opportunities for a fault-tolerant, highly available system.

2.9. TRADEOFFS: Hardware efficiency (cost per unit work) can be traded for performance and/or reliability.

3. DESCRIPTION OF THE SYSTEM
3.1. ARCHITECTURE
3.1.1. CONFIGURATIONS
3.1.1.1. INTERCONNECTIVITY: The configuration is basically a conventional multiprocessor system, but on a much larger scale than in existing systems. The structure of the system is given in Figure 1.

There are two switches, Smp and Skp. Smp allows the processor to communicate with primary memories. Skp allows the processor to communicate with the various controllers (K), which in turn manage the secondary memories (ms), and I/O devices (T). These switches are under both computer and manual control.

Each processor system is actually a complete computer with its own local primary memory and controllers for secondary memories and devices. Each processor has a Data Operations component, Dmap, for translating addresses at the processor into physical memory addresses. The local memory serves both to reduce the bandwidth requirements to the central memory and to allow completely independent operation and off-line maintenance. Below we describe some of the specific components shown in Figure 1.

* K.clock: A central clock, K.clock, allows precise time to be measured. A central time base is broadcast to all processors for local interval timing and interruption.

* Smp: This switch handles information transfers between primary memory, processors and I/O devices. The switch has ports (i.e., connections) for m busses for primary memories and p busses for processors. Up to min(m,p) simultaneous conversations are possible via the cross-point arrangement. Smp can be set under programmed control or via manual switches on an override basis to provide different configurations. The control of Smp can in principle be by any of the processors; one processor is assigned the control at any one time by manual reconfiguration.

* Mp: The shared primary memory, Mp, consists of (up to) 16 modules of (up to) 65K 16-bit words. The initial memories being used have the following relevant parameters: (1) they are core, (2) each module is 8-way interleaved, (3) access time is 250 ns and cycle time is 650 ns.

* Skp: Skp allows one or more of k Unibusses (the common bus for memory and i/o on an isolated PDP-11 system) which have several slow or fast controllers (Ks or Kf) to be connected to one of p central processors. The k Unibusses with the controllers are connected to the p processor Unibusses on a fairly long term basis. The main reasons for only allowing a long term, but switchable, connection between the k Unibusses and the processor is to avoid the problem of having to decide dynamically which of the p processors should manage a particular device. Like Smp, Skp may be controlled either programmatically or manually.

* Pc: The processing elements, Pc, are slightly modified versions of the DEC PDP-11. (The several models of the PDP-11 may be intermixed.)

* Dmap: The Dmap is a Data Operations component which takes the addresses generated in the processor and converts them to addresses to use on the Memory and Unibusses emanating from the Dmap. There are four sets of eight registers in Dmap, enabling each of eight 8 192-byte blocks to be relocated in the large physical memory. The size of the physical Mp is 2E20 words (2E21 bytes). Two bits in the processor together with the address type are used to specify which of the four sets of mapping registers is to be used.

3.1.1.2. RANGE: 1-16 memory modules with up to 256K words each (core 650 nsec cycle time), 1-16 PDP-11 processors (16 bits/word), 16x16 crossbar switch with 80x10E6 words/second capacity. A two-processor, two-memory prototype has been built to test out concepts of switch and software design.

3.1.1.3. CAPABILITY: The system should be capable of executing 3 to 15x10E6 instructions per second, depending on the PDP-11 processor model. A PDP-10 can execute roughly 3 to 15x10E5 36-bit instructions per second.

3.1.2. EXECUTIVE AND OPERATING SYSTEM
3.1.2.1. MODES, and 3.1.2.2. SOFTWARE: Although the technology of operating systems has made significant progress in the past decade, there are few systems constructed specifically for multiprocessor environments. In particular, no systems have been built to support the variety of process relations (parallel, pipeline, etc.) envisioned for C.mmp. Moreover, there is a relative lack of experience in organizing computations for parallel execution. These facts have driven the operating system design to the following conservative position.

The operating system will consist of a "kernel" and a "standard extension". The kernel will provide a set of mechanisms (tools) for building an operating system, but no policies (e.g., no scheduler, no file structure, no...). The kernel will support the (simultaneous) execution of an (almost) arbitrary number of extensions.

In considering what set of mechanisms (tools) should be provided by an operating system kernel, two commonly held views of the essential nature of an operating system are relevant:

* An operating system creates a "virtual machine" to support (user) programs by providing resources and operations not present in the underlying hardware (e.g., "files", file "read" and "write" operations, etc.).

* An operating system is a resource (virtual and physical) manager and allocator.

Note the emphasis in both views on resources, their creation, management, and operations on them. From these views we infer that an appropriate set of tools for building an operating system must provide for:
* The creation of new virtual resources;
* The "representation" of a new resource in terms of existing ones;
* The creation of operations on resources and/or their representation;
* Protection (against illegal operations on a resource), uniformly over a class of resources, as well as with regard to specific instances of a resource.

3.2. FAULT TOLERANCE
3.2.1. FAULTS TOLERATED: The ultimate goal is to be able to tolerate any fault in any unit. The system can be dynamically reconfigured via the crosspoint switch (disabling specific crosspoints) and via power switching. The detection of and recovering from failures will be a major objective. As a research vehicle, C.mmp will allow the study of fault-tolerant hardware-software interaction.

3.2.2. FAULTS NOT TOLERATED: Faults (perhaps multiple) that go undetected long enough to mutilate the majority of the copies of critical systems tables may ultimately lead to an entire system crash. Early detection and/or prevention of this class of faults will be closely studied. Multiple failures in the crossbar switch might also lead to system failure.

3.2.3. TECHNIQUES: The final hardware/software configuration for C.mmp is far from stabilized. However the following techniques either are incorporated, or provisions for incorporation have been made, or (for incremental cost) can be incorporated.

The crossbar switch is bit sliced with provision for a Hamming code on the data bits. Spare bit-plane switching or fault-masking redundancy can be employed. Switch failures appear as either a memory or processor failure. These failures can be tolerated.

Busses can function properly when a component connected to it has power removed. Memory modules are organized as banks so that a memory failure simply removes part of the memory space.

Memory and address parity. Table-driven operating systems can be written which allow graceful degradation from failure in a memory or in a processor module (removing a resource from availability). Software recalculation on multiple copies of these critical systems tables will assist failure tolerance or recovery.

Alternatively, critical computations might be performed by two distinct methods within a single processor. Diagnostic programs can be run just before critical computations are to be performed, at fixed intervals, or simply whenever the processor is not occupied with other tasks.

3.3. MODULITY: The distributed nature of operating systems allows for fault tolerance without massive expenditures for specific hardware. Software can be devised to function without faulty units. Critical calculations can easily be recalculated for checking purposes. A faulty unit is easily isolated via the crossbar switch.

An extension to the decoding process for a single error correcting/double error detecting Hamming code to enable double error correction has been investigated.

3.4. INFLUENCES: Many previous efforts have had influence on the design but there is no single major influence.

3.5. HARD-CORE: There is only one portion of the system which is not replicated--the crossbar switch. The switch has been designed so that failures appear either as a memory or a processor failure. Bit slicing, Hamming codes, and fault-masking redundancy can help to increase the switch reliability.

4. JUSTIFICATION
4.1. RELIABILITY EVALUATION: Reliability will be estimated via analysis.

4.2. COMPLETENESS: Evaluation not yet finished.
4.3. OVERHEAD: To date the hardware for fault tolerance is certainly less than 5%. However the design will evolve and quite probably raise this percentage. Software cost (in execution time) is difficult to estimate at this time.

4.4. APPLICABILITY: Any multiP crossbar configuration.
4.5. EXTENDABILITY: Syatem cannot be expanded beyond 16 memories and 16 processors without a new crossbar switch.

4.6. CRITICALITIES: Analysis shows that the selection of the memory cycle time and number of processors greatly affects system performance and cost-effectiveness. Contention in the crossbar switch limits ultimate performance.

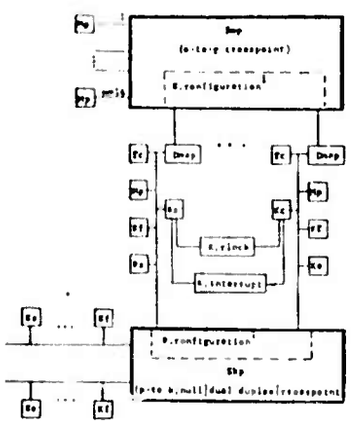
4.7. IMPLICATIONS: Programmers must ensure that their system is correct, even under conditions of asynchronous process communication.

5. CONCLUSIONS
5.1. STATUS: First portion of the hardware system should be completed by the end of Summer 1973. Portions of Hydra (the operating system) are operable.

5.2. EXPERIENCE: None to report yet.
5.3. FUTURE: In the immediate future C.mmp will be brought up as a research tool for the Computer Science Department. As a research tool it will most likely continue to evolve in design.

5.4. ADVANCES: Off-the-shelf, plug-compatible fault-tolerant (or at least self-checking) components would be very desirable. As hardware becomes cheaper, the capacity of modules become larger. And with LSI the insides of a module are not even accessible. Hence building fault-tolerant systems with off-the-shelf components without self checking or fault tolerant features is very inefficient. (Other than duplication and comparison, or triplication and voting, little else is available to the system designer.)

System validation (integrated hardware and software) is another important area. Also desirable would be a methodology for designing a fault tolerant system. Which fault tolerant techniques complement each other? Finally, switches for reconfiguration, switch control, and fault tolerant switch design are areas requiring further study.



where Pc/control processor, Pp/primary memory, T/terminals,
Pc/slow device control (e.g., for Teletype),
Pf/fast device control (e.g., for disk);
Pc/control for clock, timer, interprocessor communication

Both switches have static configuration contro. . nual and program control

Figure 1. Proposed CMU Multiminiprocessor Computer/C.mmp

SURVEY OF FAULT TOLERANT COMPUTER SYSTEMS

Oonald C. Wallace
Stanford Reaearch Institute, Menlo Park Ca, June 72

1. IDENTIFICATION
1.1 NAME:COMEX- Online order handling aystem

1.2 RESPONSIBILTY: P.C.Servlce Corp. (subsidisry Pacific Cosst Stock Exchsnge)

1.3 SUPPORT: Member firms of PCSE

1.4 PARTICIPANTS: Member firms of PCSE

1.5 START: Contract let - 17 November 1967

1.6 COMPLETION: System accepted - 4 Oecember 1969

1.7 BIBLIOGRAPHY: The moat accurate desriptlon of the COMEX Is the final documentsion deliverad with the system. Oocuments:
Specificstion for data processing and communication equipment for Pacific Coaat Stock Exchsnge PC Service Corp., 1967
Proposal for Real-Time Order Handling System BBN #p68-DE-01,4 August 1967
Contract for Real-Time Order Handling Syatem for Pacific Cosst Stock Exchsnge BBN/PCSE,17 November 1967

2. MOTIVATION
2.1 PURPOSE: Real time odd-lot order execution

2.2 PHYSICAL ENVIRONMENT: Ground baaed

2.3 COMPUTING ENVIRONMENT: The system servea two trading floors, one In Los Angeles, the other in San Francisco.

2.4 COMPUTING OBJECTIVES: COMEX Ia deaigned to handle virtually all low-speed teletype apeeds, lavals and codes. It appears as a node on each of tha connectad broker firms communication networks and must conform to the line protocols and hardware constraints of that natwork. The design objectives were for 64 "nodes" In LA. and 64 in SF., and for a maximum meaaage-switching traffic of 25,000 orders/transactions per day.

2.5 RELIABILITY OBJECTIVES: Tha system was dealgned to provide 99%+ uptime and with s no "meaaage lost" criteria.

2.6 DYNAMIC VARIABILITY: The system is deaigned ao that order entry is performed In real time, but the order execution process may lag an arbitrary period of time. In operation this lag never exceeds 20 minutes (approx.??).

2.7 PENALTIES: COMEX haa varlous degreea of degradation, the ultimate being total manual oparation and execution of the orders by the apecialists on the trading floors. Esoteric aoftware/hardware malfunctions could cauae extremly large manual Intervention problems aa the aystem is really buying and selling stock on the behalf of members of the exchsnge.

2.8 CONSTRAINTS: The PCSE Is really two exchangea with two different trading floors, one In Los Angelea and one in San Francisco. For reliability reasons the aystem Is fully redundant. A PCSE conatraint on the aystem wss that the system be equally aplit between the two aites.

3. OESCRIPTION
3.1 ARCHITECTURE
3.1.1 CONFIGURATION
3.1.1.1 INTERCONNECTIVITY: See disgram which shows the twin IBM 360 computers and the 680 aystems each of which includes s DEC POP8 computer.

3.1.1.2 RANGE: The aystem is really two syatems running In parallel. It Is sensible to run them aa aingle units or s fully redundant system. Two configurations are possible:-
Non-partitioned trading floors:
LA-remote680, SF-local680 snd SF-360
SF-remote680, LA-local680 and LA-360
Partitioned trading floors:
SF-locs1680 snd SF-360
LA-local680 and LA-360

3.1.1.3 CAPABILITY: COMEX consists of two (2) 360/50 computers plus the front-end communications systems.

3.1.2 EXECUTIVE and operating syatem: COMEX runs under IBM/360 DOS with its fixed number of multiprogram partitions option.

3.1.2.1 MODES of operation: The order execution process runs in a high priority partition of DOS while normal operation of PC Service Corp. computer operations are being run in other "foreground" snd the background partitions. The communication process (in the 680's) is dedicated and allows no other functions.

3.1.2.2 SOFTWARE organization: Basically the 680's do character assembly (bits), line protocol interpretstion (answer back, echo, etc...), messsge aegmant assembly, I/O buffering, tranamission to local and remote 360's. The 360's do message switching, code translation, meaaage decoding (syntax analysis), order queuing, decoding of NYSE and AMEX tickers (identify trades), execute queued orders, send confirmations to broker and specialist.

3.2 FAULT TOLERANCE
3.2.1 FAULTS TOLERATED: Essentially the system will tolerate sny or all failures in a single system (i.e., backup or primary).

3.2.2 FAULTS NOT TOLERATED: Any simultaneous failures in both the primary and backup system causes loss of integrity of the data files. This is considered s catsatrophic event and some manual correction and intervention for order execution and notification will be needed. (To my knowledge this has only occured once in the almost three years of operation.)

3.2.3 TECHNIQUES:
HARDWARE: The COMEX system Is completely redundant (two of everything), snd both systems run in parallel. The major design criteria was that nothing should happen in one system half that could adversly effect the other. This led to the system interconnections (PCU, being unidirectional snd step-locked in a "here's a word, take s word" fashion. All TTY connections to the system are dual dropped and there is a hardware interlock to prevent both 680 machines from outputing to s line at the same time.
SOFTWARE: The software is deaigned to be very modular, and no control flow exists between functional routines. Control flow is between the COMEX scheduler/executive and each functional module. Data is passed from function to function by means of stacks and lists, and standard system global routines are used to accomplish this. Both systems are actually performing the entire order execution task in parallel and there is really no communication between them. The only difference is that the "backup" system is not outputing transaction confirmations and order receipt notifications. The backup system maintains a queue of the last "n" messages to each line in the system. When switch-over occurs, these messages are output to the specialists/brokers with a "may be duplicate" tag.

3.3 NOVELTY: The interconnection of the OEC 680's snd the S/360's is accomplished without requiring modifications or additions to the IBM operating system or providing "special" I/O modules. The 680's (two of them) have a S/360 channel equivalent (FCU) that talks to the IBM 2841 disk controler with the two channel feature (8100). This is the equivalent of having two 360 systems talking to one disk system. This is a standard IBM configurstion possibility (though not supported hy IBM software). If the user is willing to accept implementing his own read/write lock mechanisms there is nothing in the IBM system to preclude this mode of operation. Given all of the above it is now possible to write a communications system strictly at the user level using standard IBM I/O software. Osta just "appears" on the disk and is read into the 360 and is in turn written on the disk snd just "disappears". The data from the 680's is written as s sequentially ever growing file, capturing an entire dsy's transactions. This allows "rerunning" a day's transactions in real time to find obscure bugs.

3.4 INFLUENCES: After spending several years working on modified or bastard 360 Systems and realizing the effort level to maintain these systems given the frequency of new IBM releases, it seemed insane to design s system that relied on any thing except the most rudimentsry features of the IBM monitor. The approach described has proven very successful in over three years of operation. To my knowledge no problems have been encountered due to the monitor/ Comex system interfsce.

4. JUSTIFICATION

4.1 RELIABILITY EVALUATION: The system has met and exceeded the design criteria over the last 2 years of operation.

4.3 OVERHEAD: Since the system is totally redundant, at least half the cost of the communications front end is due to reliability requirements. The reliability requirements of the system probably did not contribute significantly to the software design, and probably helped in the checkout and operational phases.

4.4 APPLICABILITY: The system has general applicability for communications and message switching systems where the base computer facility must be IBM (for what ever reasons). It offers significant cost savings when compared to an equivalent all-IBM equipment configuration. Its novel interfacing technique allows the users to concentrate on the application program and offers long-term savings in effort by not having a modified IBM operating system. The system has specific applicability to other small or moderate sized stock exchanges both U.S. and foreign.

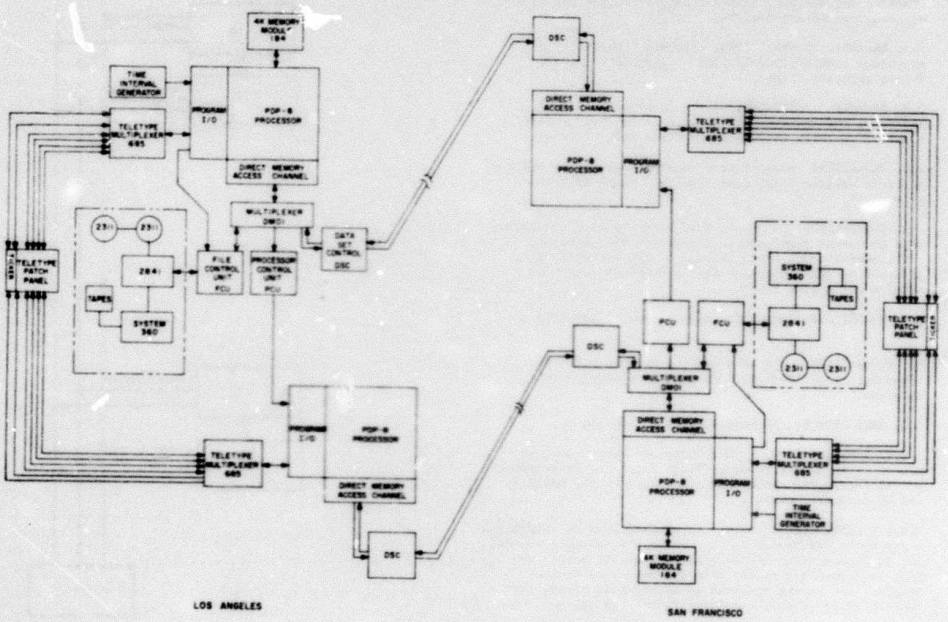
4.5 EXTENDABILITY: There appear to be no obvious extentions to the system as far as capacity is concerned. Two- or three-fold increases in throughput are possible whereas factors of ten are out of the question. Since the next obvious exchange automation task is either NYSE or AMEX and the volume of message traffic for those exchanges is staggering. COMEX most certainly has no real logical extension for these situations. Specific experience and techniques in dealing with automation of a stock exchange process may have general applicability.

4.6 CRITICALITIES: A specific goal in hardware design not to exceed the "state of the art" was imposed by PCSE to gain assurance of reliability. This constraint caused the selection of hardware that most assuredly is obsolete by today's standards (e.g., bit serial TTY interface), greatly restricting overall I/O capacity (like maybe a factor of 10).

5. CONCLUSIONS

5.1 STATUS: The system is currently handling 15% of the maximum message switching capacity of 25,000 order transactions per day. It is undergoing significant modification to handle round-lot traffic, which potentially will increase load to 50% of capacity within the next 18 months. Studies are underway to evaluate high-speed I/O capability.

5.2 EXPERIENCE: Overall system operation has been highly satisfactory to the PCSE.



COMEX SYSTEM – PACIFIC COAST STOCK EXCHANGE

SURVEY OF FAULT-TOLERANT COMPUTING SYSTEMS

John H. Wensley, Stanford Research Institute
Menlo Park, Ca. 94025, May 1972

1. IDENTIFICATION
1.1. NAME: SIFT (Software-Implamentad Fault Tolaranca), project: dasign study of a fault tolarant digital computer

1.2 RESPONSIBILITY: SRI

1.3 SUPPORT: NASA Langley

1.4. PARTICIPANTS: J. Goldbarg, K. Lavitt, R. Ratner, J. Wansley, H. Zeidlar, M. Green

1.5. START: August 1971

1.6. COMPLETION: Experimental version 1973, final dasign 1974

1.7. BIBLIOGRAPHY: Technical Progress Narratives 1-7; "SIFT - Software Implamented Fault Tolerance," FJCC 1972

2. MOTIVATION
2.1. PURPOSE: Control processing in an advanced technology transport (aircraft) including navigation, stability augmentation, engine control, instrument blind landings, etc.

2.2. PHYSICAL ENVIRONMENT: Airborne -- the system concept however is applicable to any environment.

2.3. COMPUTING ENVIRONMENT: Real-time

2.4. COMPUTING OBJECTIVES: Configuration scalability, graceful degradation, transportability of concept to any processor or memory design.

2.5. RELIABILITY OBJECTIVES: Minimum probability of erroneous results, and of loss of computing capacity during aircraft flight.

2.6. DYNAMIC VARIABILITY: Variable degree of fault tolerance for tasks of differing criticality. Ability to trade off between computing power and fault tolerance.

2.7. PENALTIES: Worst case - human lives; intermediate - aircraft damage; least case - need to abort flight objectives.

2.8. CONSTRAINTS: Hardware must be designed with weight, size and power requirements consistant with aircraft requirements. The basic concept of the system is only effected by the constraint that maintenance cannot be carried out during flight.

2.9. TRADEOFFS: Computing capacity vs. reliability

3. DESCRIPTION: A system architecture in which fault tolerance is achieved with no special fault-tolerant hardware.

3.1. ARCHITECTURE: A multi-computer (see Fig 1)

3.1.1. CONFIGURATIONS: No constraints are present on processor or memory design. Fault tolerance is achieved by the restricted connection of processors and memories, and by software control.

3.1.1.1. INTERCONNECTIVITY: Processing modules comprising a processor and memory are connected via multiple busses. The interconnection is designed so that processors may only read (and not write) into the memory of other modules. The busses are used as alternative routes rather than as multiple simultaneous transmission paths.

3.1.1.2. RANGE: The scale of the system is not frozen in the architectural concept. It is envisaged that a minimum configuration would contain three processing modules and three busses. The design does not (at present) place any limit on the maximum configuration. Greater fault tolerance is achieved with a large number of low-capability units rather than with a small number of high capability units.

3.1.1.3. CAPABILITY: The design concept is valid over the entire range of processor, memory and bus capability.

3.1.2. EXECUTIVE: Executive control (allocation, scheduling, dispatching, reconfiguration, etc.) is achieved by replicated software executive routines.

3.1.2.1. MODES: The primary operating mode is on repetitive real-time calculations involving many loosely connected tasks. Both multiprocessing and multiprogramming are included.

3.1.2.2. SOFTWARE: Tasks are multiprogrammed in each processing module. Each task for which fault tolerance is demanded is present in more than one module. A loose synchronization of task processing is achieved by the system executive (which itself is replicated and loosely synchronized). Software fault detection is carried out between each iteration of a task before erroneous results are used by the next iteration or other tasks.

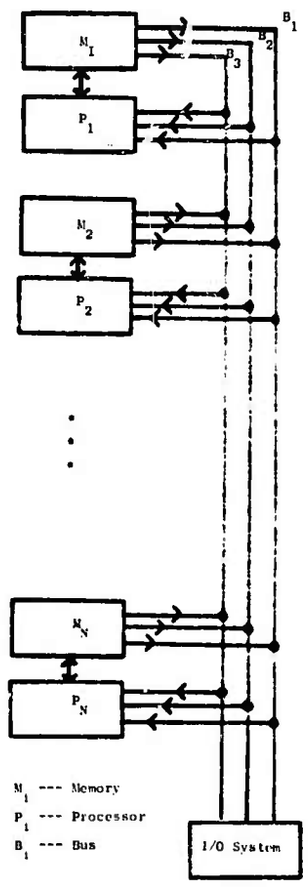


Figure 1 System Configuration

3.2 FAULT TOLERANCE

3.2.1. FAULTS TOLERATED: Tha system is tolerant to fsults in any unit (processor, bus or memory). The faults may ba tha erroneous rasult of an action (calculation, transmission or storsge) or tha failure of a unit to carry out any action.

The system handles transient, and parmanent fau ts, treating long-tarm intermittent faults as permanent. The reconfiguration procedurss can bring back into service s unit that was at one time subject to faults bu. has sinca recovered or been rapaired.

The causa of the fault (electrical, mechanical, etc.) is not of importance, the only consideration is whether the results of actions in replicated units agrae or disagree.

Independent multiple faults can be tolerated to any degree depending on the extent of replication of the function. Correlated faults both in hardware and software are not tolerated to tha same extent as uncorrelated faults. Tha loose synchronization of tasks assists in tolerating faults which are correlated in time rather than function. One-shot faults do not cause removal or reconfiguration of units from the system. The propagation of a fault from any unit to another can only occur if both units are faulty.

3.2.2. FAULTS NOT TOLERATED: Multiple correlated faults that are not detected by a voting procedure, or by repeating the task, e.g., simultaneous identical failure of two memory units when threefold replication is used. Massive faults that reduce the system to a size too small to handle the computing load.

3.2.3. TECHNIQUES: Fault detection is carried out by replication and voting. Other fault detection methods (hardware or software) are compatible with and can be incorporated into the system concept. Fault correction (or tolerance) is achieved by voting after replication in most cases but can be supplemented by other techniques such as repetition or roll-back. Tha allocation of resources to tasks can be changed either when faulty units are removed or when tha mission demands different fault tolerance and/or computational power.

3.3. NOVELTY: Lack of need for special hardware units to facilitate fault tolerance. Ability to trade off fault tolerance with computing power. Applicability of the system concept to different memory or processor designs.

3.4. INFLUENCES: The design is influenced by the need to avoid special hardware for fault tolerance, freezing fault tolerance techniques at design time, designs geared to particular size and speed computers.

3.5. HARD CORE: I don't mean anything by "hard core" in tha system described. I can imagine other system concepts in which the term has meaning (but little utility).

4. JUSTIFICATION

4.1. RELIABILITY EVALUATION: By analysis, assuming uncorrelated faults of equal probability in each part of the system (chip, connector, cable, etc.).

4.2. COMPLETENESS OF EVALUATION: Incomplete.

4.3. OVERHEAD: Variable, typically a 3-1 cost penalty is paid for fault tolerance.

4.4. APPLICABILITY: General; the design is applicable to any environment.

4.5. EXTENDABILITY: Unlimited.

4.6. CRITICALITY: Multiprocessing is critical. Multiprogramming is highly desirable (see Fig 2).

4.7. IMPLICATIONS: There are no implications on the hardware designers of processors and memories. The busses are constrained in the way units communicate. The applications' software must be implemented so that input data for a program is fetched by calling a general system routine which carries out fault detection and correction.

5. CONCLUSIONS

5.1 STATUS; A conceptual design of hardware, software and fault tolerance procedures exists.

5.2. EXPERIENCE: Software design studies show that the time and memory requirements of the fault detection and correction routines are reasonable.

5.3. FUTURE: The projection is for an experimental version of the system to be built.

5.4. ADVANCES: I/O units with fault tolerance capability.

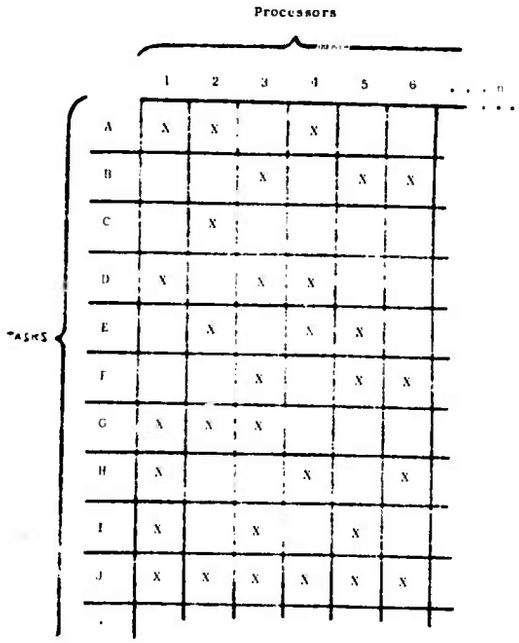


Figure 2 An Example of Task/Processor Allocation

SURVEY OF FAULT TOLERANT COMPUTING SYSTEMS

Robert K. Williams, Plessey Telecommunications Research Ltd., Taplow Nr. Maidenhead, Berks., U.K., October 1972.

1. IDENTIFICATION
1.1. NAME: System 250

1.2. RESPONSIBILITY: The Plessey Co. Ltd.

1.3. SUPPORT: System development is jointly supported by The Plessey Co. Ltd. and the National Research and Development Corporation.

1.4. PARTICIPANTS: The Plessey Co. Ltd.

1.5. START: January 1969

1.6. COMPLETION: Prototype completed end of 1971.

1.7. BIBLIOGRAPHY: The following four papers are contained in the proceedings of the International Switching Symposium, M.I.T., Cambridge, Mass., U.S.A., 6-9 June 1972.
*D. M. England, Operating System of System 250.
*J. M. Cotton, The Operational Requirements for Future Communication Control Processors.
*D. Halton, Hardware of the System 250 for Communication Control.
*W.A.C. Hemmings, Telephone Switching based on System 250.

The following five papers are contained in the proceedings of the I.E.R.E. Conf. on Computers- Systems and Technology, Middlesex Hospital Med. School, London, U.K. 24-27 Oct 72.
*R. K. Williams, System 250 - Basic Concepts.
*M. J. Goodier, System 250 - Processing Philosophy.
*P. C. Venton, System 250 - Input/Output.
*R. J. Leaman, System 250 - Security Philosophy.
*G. Edge, System 250 - Diagnostics.

The following four papers appear in the proceedings of the International Conference on Computer Communication, Washington D.C., U.S.A., 24-26 October 1972.
*D. C. Cosserat, A Capability Oriented Multi-processor System for Real-Time Applications.
*K. H. Hamer-Hodges, Fault Resistance and Recovery Within System 250.
*C. S. Repton, Reliability Assurance for System 250, A Reliable Real-Time Control System.
*J. Crompton, Structure and Internal Communications of a Telephone Control System.

2. MOTIVATION
2.1. PURPOSE: Stored program control of telephone and data switching systems.

2.2. PHYSICAL ENVIRONMENT: Ground based
2.3. COMPUTING ENVIRONMENT: The system is designed to allow flexible interaction with its environment e.g. locally, remotely and/or via a network.

2.4. COMPUTING OBJECTIVES: The computing objectives are not well defined in any absolute sense. System performance must be adequate to enable very large telephone exchanges to be adequately controlled, yet the cost of the smallest secure configuration should be minimized to allow economic control of small exchanges. The system architecture should allow easy expansion of an initial configuration by a factor of three or more whilst the system is on-line. Such expansion could be in terms of processing power and/or storage capacity and/or input/output capacity or any permutation thereof. (See also 2.8.)

2.5. RELIABILITY OBJECTIVES: The system was designed with the aim of meeting the reliability requirements proposed by the British Post Office for application to telephone control equipment. These requirements were defined on a sliding scale which related duration of a single system failure to the maximum acceptable mean frequency of occurrence of similar failures.

Failure Duration	Max. acceptable mean frequency
20ms	50 per year
15s	12 per year
2 min	1 per year
5 min	1 per 20 years
10 min	1 per 50 years

For the purposes of the above, a system failure is defined as a fault which affects more than half of the controlled equipment. /Note: Average duration 5 seconds/.

2.6. DYNAMIC VARIABILITY: Both performance and degree of fault tolerance may be varied at will by simply adding or subtracting system modules. Addition of modules simultaneously increases both performance and reliability, thus the question of trade-off does not arise.

2.7. PENALTIES: Faulty operation will obviously degrade performance which may well lead to loss of revenue and in extreme circumstances could involve loss of life e.g. if emergency telephone calls fail to get through etc.

2.8. CONSTRAINTS: There are now well defined constraints on size, weight, power, cost etc. in absolute terms. The aim has been to produce a system which is very competitive in terms of the above parameters with contemporary systems but offers very much enhanced:
* Reliability and Security,
* Ease and Range of Expansion,
* Flexibility in terms of being able to tailor the hardware and software configuration to closely match particular requirements.

2.9. TRADEOFFS: Computing capacity & reliability vs. cost.

3. DESCRIPTION
3.1. ARCHITECTURE
3.1.1. CONFIGURATIONS
3.1.1.1. INTERCONNECTIVITY (See Figure): The basic hardware constraints on system interconnectivity (aside from any additional constraints imposed by software) are described below.

Each processor unit has its own dedicated communications bus for communicating with store or the input/output network. No processors will be directly connected together under normal circumstances although this is allowed (via a special interface) for fault diagnosis purposes only. Any processor can access any storage location and any part of the input/output system. Store modules are connected to all processor buses via multiport access units.

In systems which contain more than two processors, access to the input/output system is achieved via two multiport Bus Multiplexors which multiplex three or more Processor Buses onto two Peripheral Buses (one per multiplexor). Fast peripheral devices are connected directly to both peripheral buses via two port Parallel Interface Units. All data transfers between the above units take place in 24 bit parallel mode.

Slow speed and/or low activity peripheral devices are connected to the system via a serial communications medium in which all data transfers take place in serial bit form. The Serial Medium is interfaced onto the Peripheral Buses via specialized Parallel Interface Units known as Serial-Parallel Adaptors. Each adaptor has two ports and is connected to both Peripheral Buses. Peripherals are interfaced onto the Serial Medium via two part Serial Interface Units, each port being connected to a different Serial-Parallel Adapstor via a network of Data Switches. The pathway between a Serial-Parallel Adaptor and a Serial Interface Unit normally passes through a 64 port Primary Data Switch (of which there is one per Serial-Parallel Adaptor) and then through a 16-port Secondary Data Switch to the appropriate Serial interface unit. The secondary Data Switch may sometimes be omitted.

Large systems may contain several Serial Media each being connected onto two Peripheral Buses via two Serial-Parallel Adaptors. If necessary several pairs of Peripheral Buses could also be provided via several pairs of Multiplexors.

If there are no more than two processors in a system, Multiplexors are unnecessary and Parallel Interface Units and Serial-Parallel Adapstors may be connected directly to the processor buses.

3.1.1.2. RANGE: There are no well defined upper limits on the numbers of processors and/or stores possible in a system, but present estimates indicate that systems containing up to 16 processors and perhaps 20-30 store modules are feasible. Each store module could contain up to 64K of 24 bit words. The smallest possible system currently envisaged would contain a single processor and a single store module of 16K or 24K capacity.

3.1.1.3. CAPABILITY: Based on a method of power assessment which has been developed specifically for the telephone switching application, a single processor system turns out to be about one third or one half (depending on the type of store used) as powerful as an IBM 360/65. The maximum

possible number of fixed point additions per second for a single PP250 processor lies between about 500,000 and 900,000 depending on the type of store used (viz. 850ns core or 300ns plated wire) and on whether the additions is a store reference or register to register operation.

3.1.2.1. MODES: The system is a multi-CPU system with all CPU's being asynchronous identical units of equal status.

All Operating System modules are re-entrant and thus may be executed by several processors simultaneously and independently. The Operating System will normally be entered by a subroutine call but can also be entered as a result of a program trap or as the result of a timer maturing within a CPU.

The System is multiprogrammable with each processor being run on a time-sharing basis. Multi-processing is a standard feature of the System and processes can be run independently from or in controlled co-operation with other processes.

In System 250 a process is a dynamic entity and is defined as the execution of program code on a particular set of input data. Because of the protection afforded by Capabilities, many processes can safely share a particular block of program code simultaneously, but each will execute it on a different set of data. A process may only be run on one processor at a time but may in general run on several different processors consecutively.

Since it is data and not code which distinguishes one process from another, processes are allowed to cross the conceptual boundary between Operating System and user programs in just the same way as they would cross the boundaries between individual user programs. This presents no special difficulties since the hardware Capability mechanism which monitors and constraints the switching of control between programs, makes no distinction between Operating System and user programs.

3.1.2.2. SOFTWARE: The System 250 software organization is described in D.M. England's paper presented at the International Switching Symposium, June 1972. (See 1.7.)

3.2. FAULT TOLERANCE

3.2.1. FAULTS TOLERATED: The System 250 architecture allows at least one redundant module of each type to be provided in a system. Thus the system will carry on operating in the face of hardware faults provided that at least one module of each type remains fault-free. Faults caused by software errors will normally only occur (assuming programs have been properly debugged) when rather rare combinations of data and/or timing are encountered. The software recovery procedures outlined in 3.2.3 below allow the unusual circumstances surrounding the fault to be avoided by employing increasingly powerful (and hence more disruptive) recovery actions until the fault no longer manifests itself.

It is recognised that a number of obscure software errors are always likely to be present in the system but since the circumstances, which cause system faults to develop as a result of these errors, are by definition rarely encountered, they will not in general cause unacceptable service disruption.

The effect of a hardware or software fault on the external environment will be to cause one or more of the following:

* If the fault disables a store or processor, a permanent drop in the throughput of the system will result, at least until the necessary maintenance action is undertaken.

* If the fault is elsewhere a temporary fall in the throughput capacity of the System will occur while test and restart or reload measures are undertaken. The magnitude and duration of this fall depends on the type of fault, the status of the System (i.e. with regard to work load) and the hardware and software configuration of the System.

* Depending on the nature of the fault, it may be possible to restart affected processes at the point at which the fault was detected or it may be necessary to restart processes from the beginning. In the telecommunications control application the former action should cause no loss of calls whereas the latter action may mean the loss of some or all of the calls being handled by the affected processes. In the worst case the whole system is reloaded from backing Store and all read/write data areas are cleared resulting in the loss of all calls being handled by the system. This case should be very rarely encountered.

* Faults in Serial or Parallel interface Units will naturally disable the peripherals to which they are attached. These units are allocated on a one per peripheral basis thus a single fault will only affect one peripheral device. All communication paths between processor and peripheral interface units are duplicated thus a fault in one or more of the units on only one of the communication paths will not affect system operation.

3.2.2. FAULTS NOT TOLERATED: it is anticipated that the only fault conditions not tolerated by the System (i.e. from which the system is unable to recocover automatically) involve at least two simultaneous faults which

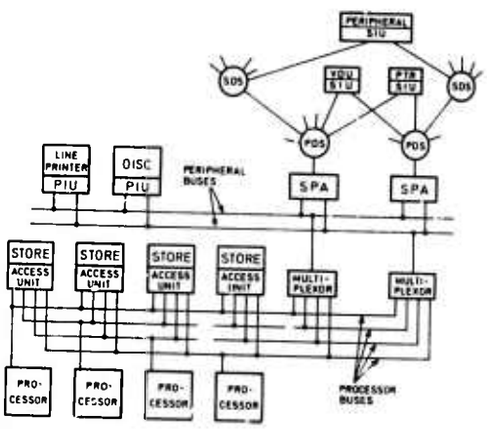
* Disable at least two system hardware modules of the same type so that no fault free modules of this type remain, or
* Overide the Capability mechanism and corrupt ALL copies of a vital software area before the fault is detected.

It is believed that the chances of either of the above happening are acceptably remote. In any specific application the chances of such situations arising can always be reduced below any finite limit by suitably increasing redundancy of hardware and software modules.

3.2.3. TECHNIQUES: The System 250 architecture allows a fault to be tolerated in any single hardware module by providing redundant modules of each type in a secure system configuration. Faults will normally be detected either by one of an extensive range of hardware fault detection mechanisms provided in each PP250 processor unit (e.g. capability checks, parity checks, microprogram checks, etc.) or by background test routines or by consistency checks written into the Operating System and applications software.

Faults detected by hardware automatically cause the processor concerned to enter a self-test routine with very limited access to system resources. Processors which succesfully emerge from the self-test can apply to rejoin the system, the application normally being dealt with by fault recovery software being run on a good processor. Processors which have a bad history of faults may be refused permission to rejoin the system and forced to endlessly repeat the self-test procedure until maintenance action is undertaken.

if a hardware fault is traced to a module other than a processor the fault recovery software causes the faulty module to be effectively isolated from the system awaiting maintenance action. if (as may be the case for certain intermittent hardware faults) the fault cannot be traced to a particular module the recovery software will as a last resort cause the system to be reconfigured leaving out modules on a trial basis, until a fault free configuration is achieved.



EXAMPLE OF MEDIUM INSTALLATION

Software recovery procedures are arranged in a hierarchical structure starting with procedures which attempt to clear corruptad data and restart failed processes and if necessary working up through several stages of successively more disruptive recovery measures each involving reloading code and data areas from copies on backing store. Eventually, if all else fails, this culminates in clearing all code and data areas from fast store (except certain areas containing replicated copies of the basic recovery programs), clearing all read/write data from backing store and reloading all programs and read/only data from backing store.

Recovery from both hardware and software faults involve the implementation of a series of increasingly disruptive actions until the fault is cleared. When one processor in a multi-processor system becomes faulty and enters its self-test routine, its absence is soon noticed by a System Monitor process which inspects the status of the system and is scheduled to run at regular intervals. This process initiates a high priority recovery process which is then placed in a 'ready to run' list and in due course is scheduled to run on a good processor just like any other process.

3.3. NOVELTY: The most unusual design features of the system are as follows:
* The system has been designed to be fault tolerant to a degree hitherto unheard of in commercially available computer systems.
* System power and storage may be expanded independently simply by adding further processors or storage units.
* Additional "redundant" Processors and stores added to a system to enhance its reliability, also perform useful work and thus increase the computing capacity of the system. These modules are therefore not redundant in the same sense as redundant modules in many other systems which purely perform a backup function and do not usefully contribute to system performance in the absence of a fault.
* System hardware and/or software modules can be inserted, removed and/or modified whilst the system in on-line with no consequent loss of service.
* Data and program security is preserved by a hardware implemented Capability mechanism which not only defines the areas of store or the input/output system which are accessible by a program, but also defines the type of access allowed. This is a particularly important feature when the sharing of store is allowed between processes.
* There is no privileged mode of processor operation. Operating System programs are subject to the same security restrictions (enforced by Capabilities) as user programs.
* In the event of a fault being detected in hardware, a hierarchy of automatic recovery procedures is entered with, if necessary, successively more disruptive measures being taken in order to recover a working system. This leads to a trial reconfiguration procedure if all else fails.
* Diagnosis of a faulty hardware module may be carried out on-line with no increased risk to the rest of the system.
* The input/output system is designed to be very flexible in its configurability and in particular allows very large numbers of low activity peripheral devices to be efficiently dealt with.
* No external interrupts are allowed into the processors (for security reasons) and all input/output is handled via polling procedures.
* Virtual memory is used in a real-time context.

3.4. INFLUENCES: The use of Capabilities to structure and protect the System 250 software has been significantly influenced by the research work of Dr. R. S. Fabry carried out at the University of Chicago on "List Structured Addressing" and also by the ideas and advice of Professor M. V. Wilkes of the University of Cambridge.

3.5. HARDCORE: The PP250 processor has been designed such that the conventional conception of hardcore (i.e. that single portion of the system which must work in order to make the system work or make diagnosis possible) has been avoided. Replication of all vital system hardware and software modules ensures that no single module failure can bring the system down.

4. JUSTIFICATION
4.1. RELIABILITY EVALUATION: System reliability calculations have been carried out using estimated M.T.B.F's and M.T.T.R's of system hardware modules. These in turn were calculated from measured failure rates of individual hardware components. The processor self-test program has been tested using a logic level simulation program for the processor into which a large number and variety of faults were injected.

4.2. COMPLETENESS OF EVALUATION: The design evaluation is expected to continue for some considerable time (if indeed it ever stops) especially in the light of running Operating System (well under way) and applications programs.

4.3. OVERHEAD: This depends very much on the required system power and the required level of reliability. In the smallest secure configuration in which all modules are duplicated one could argue that more than 50% of the cost is devoted to the provision of fault tolerance. However, even in this case the extra processor and extra store make real contributions to system performance and so cushion system against instantaneous peaks. See also 3.3, item 3.

In large systems the ratio of essential to "redundant" hardware may be greater than 5 to 1 depending on system size and the desired level of reliability.

The proportion of fast storage devoted to fault recovery software in a typical telecommunications application will probably be not more than 25% and could be a lot less in a large system. Probably more than 50% of backing storage however is present in order to achieve fault tolerance since backing storage containing copies of all system software must be duplicated for reliability.

It is difficult to assess the cost overhead associated with the use of capabilities since their usefulness extends far beyond just fault protection.

4.4. APPLICABILITY: The system is applicable to almost any real-time control application but particularly those with a good reliability and expansion potential.

4.5. EXTENDABILITY: This question cannot be satisfactorily answered at this a age as it requires a much more complete evaluation of the present system design.

4.6. CRITICALITIES: Both multi-programming and multiprocessing are fundamental to the achievement of the system design aims. The choice of all except peripheral and storage hardware is critical as all other system hardware modules have built in features which are closely matched to the overall system requirements. It is of course possible to use modules which offer the same facilities and interfaces but differ internally in detailed implementation.

4.7. IMPLICATIONS: Main requirement on hardware system designers is that design should not allow single hardware failures to generate further failures and thus spread to several modules. Software designers are responsible for inserting consistency checks, etc., in their own programs. They should also write routines which enable execution of their programs to be restarted following a detected fault. This responsibility also extends to user programmers. Maintenance action should be designed such that it can be carried out on-line.

5. CONCLUSIONS
5.1. STATUS: Several pre-production multiprocessor systems working and under evaluation. Production expected to commence in middle of 1973. First delivered production system expected to be fully operational in September 1974.

5.2. EXPERIENCE: The basic system philosophy is a proven success. Planned development targets are being consistently achieved. Some minor modifications are being introduced as a result of evaluation of a number of pre-production systems.

5.3. FUTURE: As a general policy system implementation is continually under critical review in the light of operating experience and advances in technology. The ability to allow system evolution is essential in applications such as telecommunications control where the system is designed to operate continuously for perhaps several decades.

5.4. ADVANCES: In respect of system architecture, so many novel features are incorporated in the present System 250 design that these require a more complete evaluation before all of the important implications become apparent. It is therefore not possible at this stage to indicate architectural advances which are obviously desirable.

Obviously desirable advances in hardware technology include increased reliability of peripherals and minimum use of moving part mechanical techniques in particular.

possible number of fixed point additions per second for a single FP250 processor lies between about 500,000 and 900,000 depending on the type of store used (viz. 850ns core or 300ns plated wire) and on whether the additions is a store reference or register to register operation.

3.1.2.1. MODES: The system is a multi-CPU system with all CPU's being asynchronous identical units of equal status.

All Operating System modules are re-entrant and thus may be executed by several processors simultaneously and independently. The Operating System will normally be entered by a subroutine call but can also be entered as a result of a program trap or as the result of a timer maturing within a CPU.

The System is multiprogrammable with each processor being run on a time-sharing basis. Multi-processing is a standard feature of the System and processes can be run independently from or in controlled co-operation with other processes.

In System 250 a process is a dynamic entity and is defined as the execution of program code on a particular set of input data. Because of the protection afforded by Capabilities, many processes can safely share a particular block of program code simultaneously, but each will execute it on a different set of data. A process may only be run on one processor at a time but may in general run on several different processors consecutively.

Since it is data and not code which distinguishes one process from another, processes are allowed to cross the conceptual boundary between Operating System and user programs in just the same way as they would cross the boundaries between individual user programs. This presents no special difficulties since the hardware Capability mechanism which monitors and constrains the switching of control between programs, makes no distinction between Operating System and user programs.

3.1.2.2. SOFTWARE: The System 250 software organisation is described in D.M. England's paper presented at the International Switching Symposium, June 1972. (See 1.7.)

3.2. FAULT TOLERANCE

3.2.1. FAULTS TOLERATED: The System 250 architecture allows at least one redundant module of each type to be provided in a system. Thus the system will carry on operating in the face of hardware faults provided that at least one module of each type remains fault-free. Faults caused by software errors will normally only occur (assuming programs have been properly debugged) when rather rare combinations of data and/or timing are encountered. The software recovery procedures outlined in 3.2.3 below allow the unusual circumstances surrounding the fault to be avoided by employing increasingly powerful (and hence more disruptive) recovery actions until the fault no longer manifests itself.

It is recognised that a number of obscure software errors are always likely to be present in the system but since the circumstances, which cause system faults to develop as a result of these errors, are by definition rarely encountered, they will not in general cause unacceptable service disruption.

The effect of a hardware or software fault on the external environment will be to cause one or more of the following:

* If the fault disables a store or processor, a permanent drop in the throughput of the system will result, at least until the necessary maintenance action is undertaken.

* If the fault is elsewhere a temporary fall in the throughput capacity of the System will occur while test and restart or related measures are undertaken. The magnitude and duration of this fall depends on the type of fault, the status of the System (i.e. with regard to work load) and the hardware and software configuration of the System.

* Depending on the nature of the fault, it may be possible to restart affected processes at the point at which the fault was detected or it may be necessary to restart processes from the beginning. In the telecommunications control application the former action should cause no loss of calls whereas the latter action may mean the loss of some or all of the calls being handled by the affected processes. In the worst case the whole system is reloaded from backing Store and all read/write data areas are cleared resulting in the loss of all calls being handled by the system. This case should be very rarely encountered.

* Faults in Serial or Parallel Interface Units will naturally disable the peripherals to which they are attached. These units are allocated on a one per peripheral basis thus a single fault will only affect one peripheral device. All communication paths between processor and peripheral interface units are duplicated thus a fault in one or more of the units or only one of the communication paths will not affect system operation.

3.2.2. FAULTS NOT TOLERATED: It is anticipated that the only fault conditions not tolerated by the System (i.e. from which the system is unable to recover automatically) involve at least two simultaneous faults which

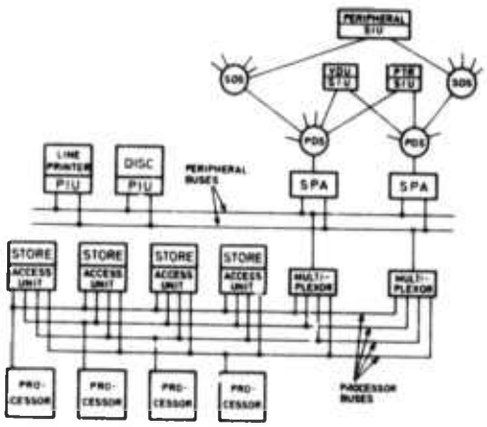
* Disable at least two system hardware modules of the same type so that no fault free modules of this type remain, or
* Override the Capability mechanism and corrupt ALL copies of a vital software area before the fault is detected.

It is believed that the chances of either of the above happening are acceptably remote. In any specific application the chances of such situations arising can always be reduced below any finite limit by suitably increasing redundancy of hardware and software modules.

3.2.3. TECHNIQUES: The System 250 architecture allows a fault to be tolerated in any single hardware module by providing redundant modules of each type in a secure system configuration. Faults will normally be detected either by one of an extensive range of hardware fault detection mechanisms provided in each FP250 processor unit (e.g. capability checks, parity checks, microprogram checks, etc.) or by background test routines or by consistency checks written into the Operating System and application software.

Faults detected by hardware automatically cause the processor concerned to enter a self-test routine with very limited access to system resources. Processors which successfully emerge from the self-test can apply to rejoin the system, the application normally being dealt with by fault recovery software being run on a good processor. Processors which have a bad history of faults may be refused permission to rejoin the system and forced to endlessly repeat the self-test procedure until maintenance action is undertaken.

If a hardware fault is traced to a module other than a processor the fault recovery software causes the faulty module to be effectively isolated from the system awaiting maintenance action. If (as may be the case for certain intermittent hardware faults) the fault cannot be traced to a particular module the recovery software will as a last resort cause the system to be reconfigured leaving out modules on a trial basis, until a fault free configuration is achieved.



EXAMPLE OF MEDIUM INSTALLATION

APPENDIX 3

DETAILED CONSIDERATIONS OF MEMORY RECONFIGURATION

This appendix considers detailed aspects of reconfiguration of memory systems, not only of the memory circuits themselves, but of other components such as data busses, address decoders, etc.

The memory is assumed to be built from a number of units -- for example LSI chips, each having the same memory capacity. When a fault is detected, some of these units are discarded, and either they are replaced by a similar number of spare units, or the system now has reduced memory capacity. The terms used are as defined in section 6.

A3.1. MEMORY RECONFIGURATION BY BLOCK REPLACEMENT

We restate the reliability estimates previously given in section 4.2.

$$P[\geq w' : w] = \sum_{i=0}^{\lfloor (w-w')/y \rfloor} C_i \, P_{fi} \qquad \text{(A1)}$$

where

$$P_{fi} = \binom{u}{i} P_f^i (1-P_f)^{(u-i)} \qquad \text{(A2)}$$

and

$$u = w/y \qquad \text{(A3)}$$

A3.2 THE USE OF CODING WITH BLOCK REPLACEMENT

The questions addressed in this section are: What is the optimum number b of bits per byte? and, given the probability p of chip failure, what is the value for $P[\geq w' : w]$

If w' words are required to be still available after t blocks have become faulty, then W=(w'+yt). The number of chips required is therefore

$$N = (k+r)(w'+yt)/(yb) \tag{A4}$$

Let N(b,t) be the number of chips required for varying b and t for the case w' = 16k, yb = 4k, n = 32. Then a few important values of N are

$$N(1, t) = 152 + 38t$$
$$N(2, t) = 152 + 19t$$
$$N(4, t) = 160 + 10t$$
$$N(8, t) = 192 + 6t.$$

There are good reasons for b to be a power of 2, although codes of course exist for other values of b (see Section 4.1). The reliability can be expressed as

$$P[\geq w' : w] = \sum_{i=0}^{(w-w')/y} \binom{w/y}{i} P_f^i (1-P_f)^{(w/y-i)} \tag{A5}$$

where

$$P_f = 1-(1-p)^d, \tag{A6}$$

whence

$$P_s = P[\geq w' : w] = \sum_{i=0}^{(w-w')/y} \binom{w/y}{i} (1-(1-P)^{(k+p)/r})(1-P)^{(k+p)(w/y-i)/b} \tag{A7}$$

Figures A3.1(a) to (e) show P_s or $P_f = 1-P_s$, i.e., the probability of success or failure for $p=10^{-n}$, n=1...5, and k=32, b=1,2,4,8 and w=16k.

A3.3. RECONFIGURATION BY CHIP REPLACEMENT

A3.3.1.THE MEMORY MODEL

The basic model is depicted in Figure A3.2. It consists of a decoder (for high-order address bits), an input switching network, an output switching network, and a set of memory chips. Each memory chip acts as a y-word by b-bit RAM. The following parameters describe the configuration of the main memory:

Fig. A3.1 Probability of success (P_s) or failure (P_f) as a function of number of chips for

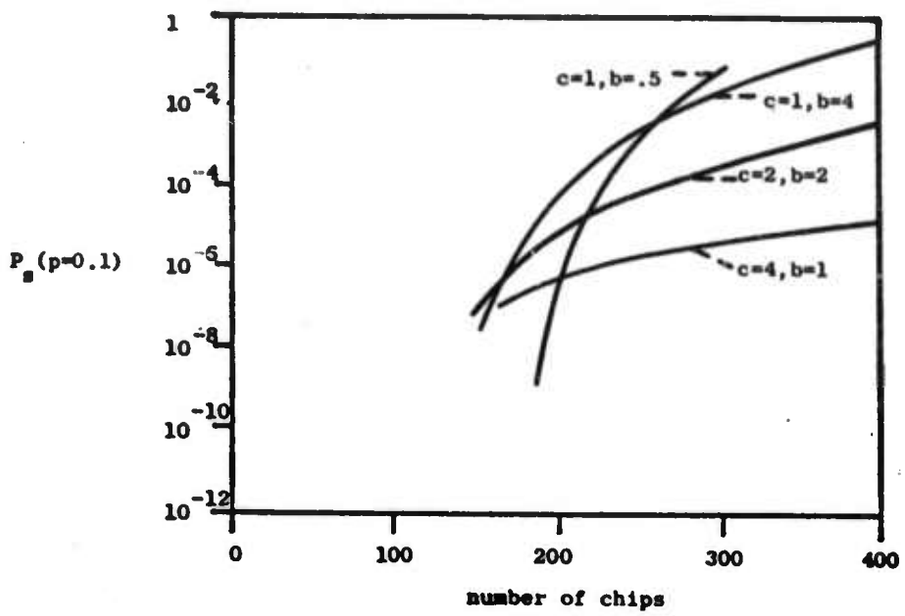
(a) p = 0.1

(b) p = 0.01

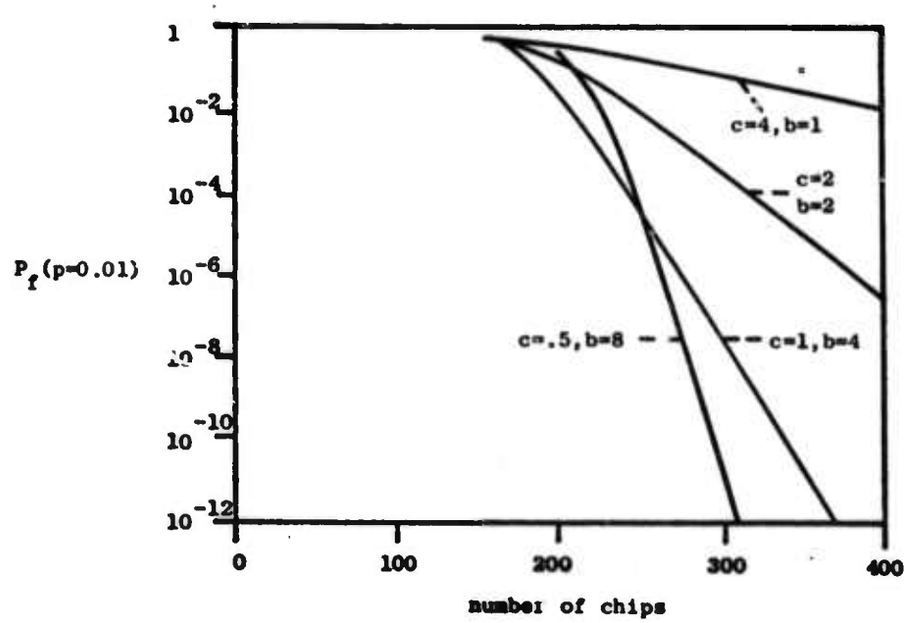
(c) p = 0.001

(d) p = 0.000,1

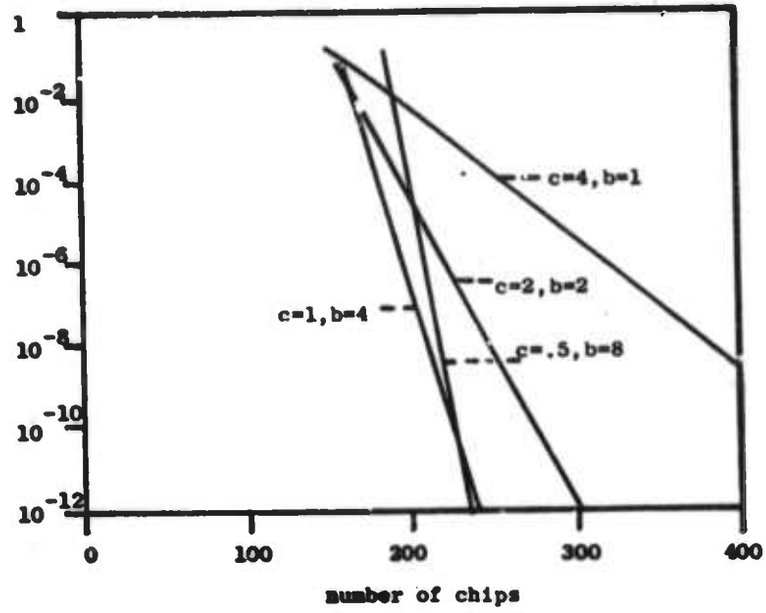
(e) p = 0.000,01



a

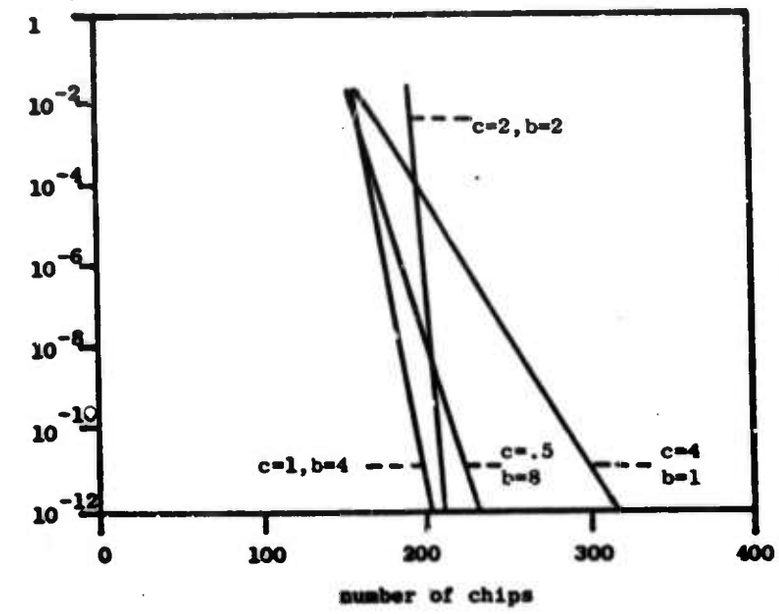


b



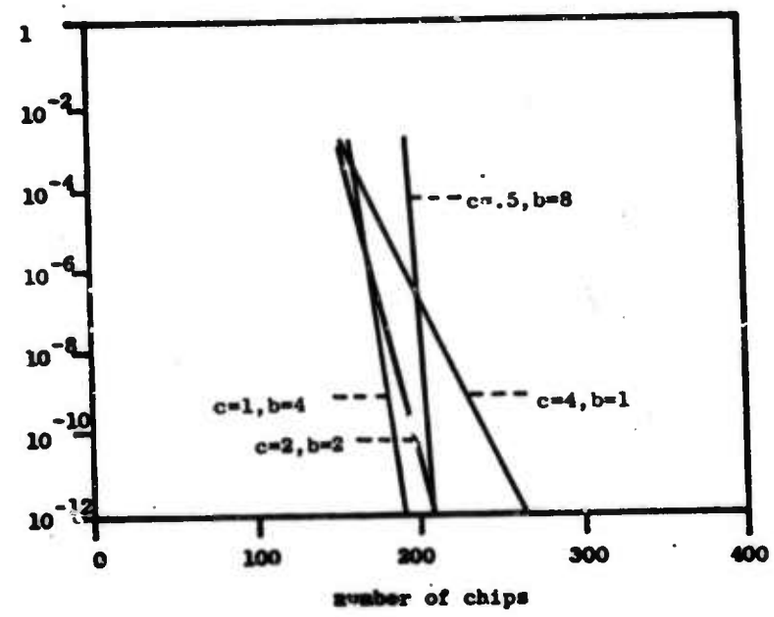
$P_f(p=0.001)$

c



$P_f(p=0.000,1)$

d



$P_f(p=0.000,01)$

e

d = number of bytes per n-bit word (typically 4 to 16), d=n/b

z= number of blocks of memory (typically 4 to 2048), where a block consists of y words

s = number of spare chips (typically small relative to total memory size)

m = total number of chips = (zd+s)

t = number of faulty chips to be tolerated.

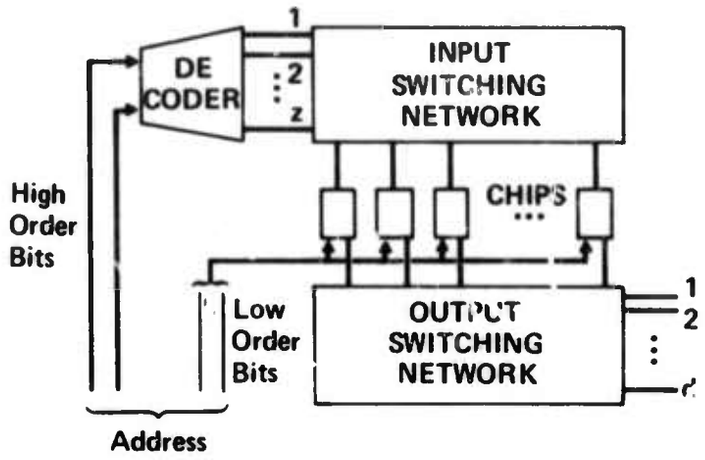


Fig. A3.2 General model for reconfigurable memory.

From the standpoint of maximal use of spare chips, s=t is desirable; however, as seen below, some benefits accrue from having s>t in terms of switching-network regularity and simplicity, and ease of switch set-up. In this model, t is the guaranteed fault-tolerance, i.e., the memory can be configured into z blocks of d chips in the presence of all combinations of t or fewer memory chip failures. In one of the examples below, s t. In this case there are sufficient spares to correct more than t failures, but switching-network limitations may prevent this extended correction. However, an analysis shows that the number of such offensive combinations is vanishingly small, and that certain economies

in switching-network complexity are attained by keeping the guaranteed correction below the number of spares. Thus, in any event the value t itself is not sufficient to evaluate the reliability of the memory.

The memory function is to be configured out of a set of zd operative chips. The block selection is accomplished by the decoder, which selects one appropriate control line, under control of the $\log_2 z$ higher order address bits. The lower-order address bits are delivered to all memory chips, with the word selection accomplished by a decoder within each chip. The appropriate configuration is achieved by setting up the input and output switching-network pairs (SNP). Note that the connection established by the SNPs needs to be modified only when the memory is reconfigured.

For most of this section, only single-level incomplete cross-bar arrays are considered. Note that in contrast with the telephone cross-bar arrays, the switching networks for the memory organization require switches at comparatively few cross-points.

As a better illustration of the role of the SNPs, consider a simple example for which z=4, d=3, s=6, m=18, t=5. Figure A3.3 displays one possible set-up of the switching networks to accommodate the indicated faulty chips. Each utilized chip is identified according to its place in memory; that is, for a chip at (i,j), "i" signifies the block and "j" signifies the byte. The activation of a particular block of d chips is accomplished by activating the appropriate control line. This activation signal is transferred through the input switching network to a unique set of d chips. The memory word emerging from the d chips is transferred to a unique set of d output data lines by the output switching network.

As noted in the next section, this example illustrates a nonseparable switching network-pair that is an SNP for which the set-up at the input and output networks must be accomplished together. For a separable network pair the set-up of one of the two networks can be done first in its entirety, independently of the other.

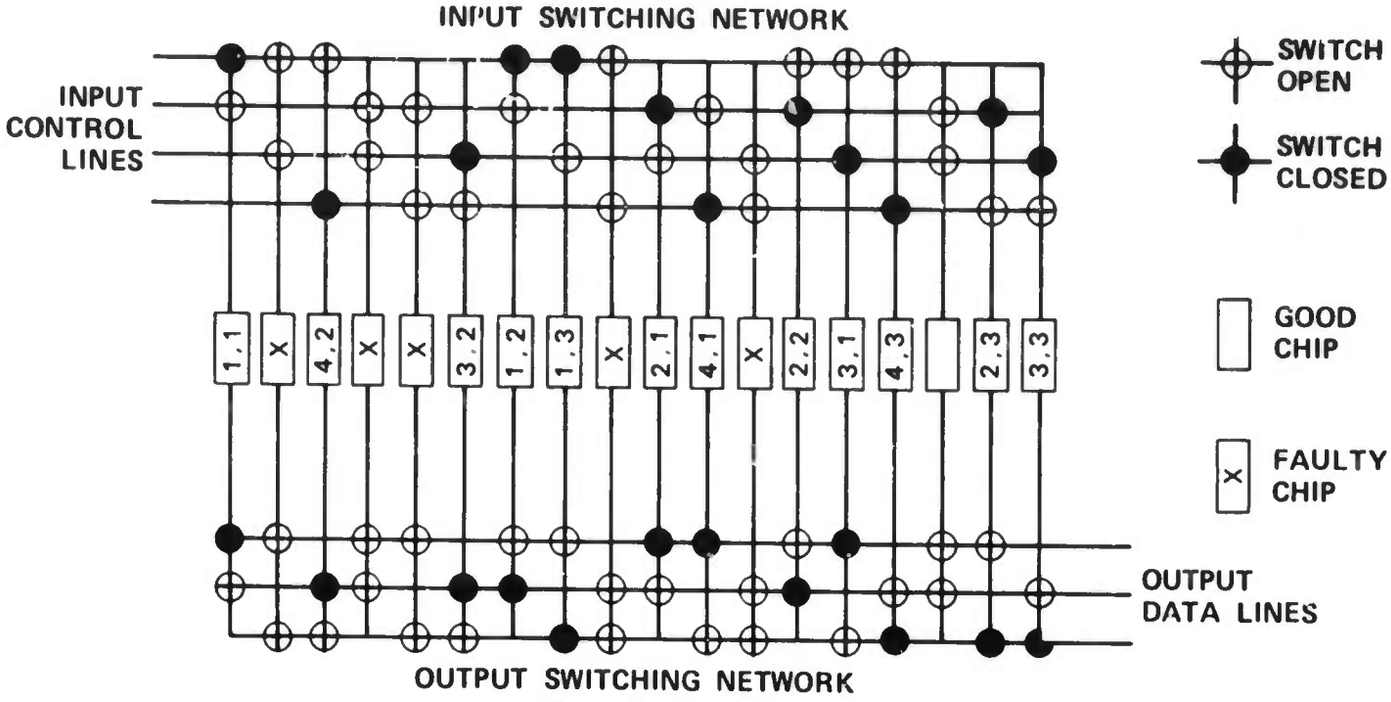


Fig. A3.3 An example chip reconfigurable memory.

Before embarking on the details of the synthesis procedures for these SNPs, it is worthwhile to indicate the possible benefits of this organization, as compared with other fault-tolerant memory organizations. Consider a modest-sized memory requirement of 32 kilowords, each 32 bit long. Such a requirement can be achieved with 16 blocks, each of which contains 16 2-bit-wide chips, for a total of 256 chips. Assuming a chip failure probability of 10^{-6} per hour, in a mission of five years ten failures might be expected. For the organization discussed in this section, a tolerance of 10 failures requires a redundancy of 10 chips, or under 4% redundancy. This can be contrasted with a memory system wherein an entire block is replaced upon the occurrence of any chip failure within the block. For this latter system to achieve comparable reliability, a redundancy of about 50% is required. Two comments are in order here. First, the lower redundancy measure is meaningful only if the switching overhead is small--a situation that we will now show to be the case. Second, chip replacement becomes more favorable as b is decreased and y increased

(with yb held constant). Moreover, if error detecting and correcting techniques are used in addition to replacement, then the smaller byte sizes are preferable from the standpoint of lower code redundancy (and simpler decoding circuitry).

A3.3.2. SWITCHING NETWORK SYNTHESIS TECHNIQUES

In this section we are primarily concerned with establishing conditions for the existence of suitable single-level cross-bar switching networks. The last subsection below deviates from this single-level formulation, to indicate a less costly multi-level network that handles large values of t.

It will be convenient to view the input network as described by the z by m matrix SI, and the output network, by the d by m matrix SO. A "1" in a particular location (e,f) of the matrix corresponds to a switch in row e and column f of the network. The following theorem gives necessary and sufficient conditions for the matrices SI and SO such that the SNP is capable of reconfiguring the memory in the presence of any t or fewer failed memory chips.

THEOREM 1: For the single-level incomplete cross-bar input and output switching networks, there exists a setting of the switches such that in the presence of t or fewer memory chip failures, the operative chips can be configured into an array of z rows by d columns, as long as the union of each combination of i rows of SI, i=1, 2, ..., z, and the union of each combination of j rows of SO, j=1, 2, ..., d, overlap in at least ij+t places.

The proof is an extension of the Diversity Theorem (Ore 63), which gives necessary and sufficient conditions for the assignment of workers to jobs.

The necessity part is obvious since for some set of i rows of the input network and j rows of the output network, there must be ij paths when the networks are configured. Since up to t chip faults are to be

tolerated, wherein each chip failure disables a path, and since a path corresponds to the appearance of 1's in a column of SI and SO , the necessity part is seen. The sufficiency part will be proven by strong induction on i,j.

(a) The sufficiency part is trivially true for i=j=1, since an overlap of t+1 places between a row of SI and a row of SO guarantees at least one good path for t or fewer failures.

(b) Define an ordered row-pair (α, β) as consisting of row α of SI and row β of SO. Define a N-1 ordered row-pair set (or simply N-1 set for short) as a set of N-1 such ordered row-pairs. The intention here is that if (α, β) is in an ordered row-pair set, then there exists a path between row α of the input network and row β of the output network. Now assume that if the conditions of Theorem 1 are satisfied for all N-1 ordered row-pair sets, then an appropriate setting of the switches can be achieved to establish paths (α, β) for all contained in the set. Note that the theorem condition, abstracted for the N-1 ordered row-pair set, is that

$$\left|\bigcup_{\forall(\alpha,\beta)}\right| P_{\alpha\beta} \geq N-1+t \ , \quad \text{contained in M subset} \tag{A8}$$

where $P_{\alpha\beta}$ is the set of overlap positions between row α of SI and row β of SO.

By $\left|\bigcup_{\forall(\alpha,\beta)}\right|$ (overlap between row α of SI and row β of SO), we mean the number of distinct columns for which there is a "1" in position (α,ξ) of SI and (ξ,β) of SO , taken over all (α,β). The "union" operation signifies that we count a column only once no matter how many times it appears because of distinct (α,β).

(c) As the induction step, we will show that given condition (b) above and the premise of the theorem for all N-1 ordered row-pair sets, then the theorem is true for all N ordered row-pair sets. There are two cases to consider. In the first case assume that for a particular set of t or fewer chips, all M subsets M<N of the N ordered row-pair set

satisfy the conditions of the theorem with room to spare. That is:

$$\sum_{\substack{|U| \\ \forall(\alpha,\beta) \\ \text{contained in M subset}}} P_{\alpha\beta} > M \ . \tag{A9}$$

Then for any (α, β) in the N ordered row-pair set, make an arbitrary switch setting to establish the path between α and β. After removing (α, β), what is left is an N-1 ordered row-pair set, and this set satisfies the conditions of the Theorem as in (b) above, thus establishing the induction step. In order to see that the theorem conditions are satisfied after removing (α, β), note that any N-1 subset left will have lost a maximum of "1" from the summation of overlaps--namely that corresponding to the column . Thus after removing the t or fewer columns in error and the column caused by the "removed" (α,β), we find

$$\sum_{\substack{|U| \\ \forall(\alpha,\beta) \\ \text{in N-1 subset}}} P_{\alpha\beta} \geq N-1 \ . \tag{A10}$$

In the second case, assume that for a particular set of t or fewer chip failures, after removing all paths through the failed chips, there is at least one subset that satisfies the theorem conditions exactly, that is:

$$\sum_{\substack{|U| \\ \forall(\alpha,\beta) \\ \text{in M subset}}} P_{\alpha\beta} = M \tag{A11}$$

If we then assign the appropriate paths for all (α, β) contained in the M subset, which we know we can do by virtue of (b) above, then for the (α, β) ordered row-pairs in the complement set, we have

$$\sum_{\substack{|U| \\ \forall(\alpha,\beta) \\ \text{in N-M subset}}} P_{\alpha\beta} \geq N-M \tag{A12}$$

since the entire N set satisfied the theorem conditions. Thus by virtue of (b), path assignments can be made for the N-M subset. The theorem is then established for an arbitrary N ordered row-pair set, so that it is certainly satisfied for a set composed of bd elements, namely, b rows by d rows.

In a subsection below we illustrate SNPs that satisfy the conditions of this theorem. The one disadvantage of these nonseparable SNPs is that the switch-setting algorithm must deal with both the input and output network simultaneously. The situation is improved with the separable networks defined next.

DEFINITION: A switching network pair is SEPARABLE with respect to the input network if the switches can be set to achieve the configuration of the memory into z rows and d columns, in the presence of t or fewer failures, and if the appropriate settings of the input network can be decided without knowledge of the output network. The settings of output network switches are then decided after those of the input network. (Separability with respect to the output network can be similarly defined, although no advantage seems to be found in such SNPs.)

The following theorem gives necessary and sufficient conditions on the SI and SO matrices for the existence of such a separable network.

THEOREM 2: An SNP, composed of single-level input and output networks, is separable with respect to the input network if and only if (iff) the corresponding SI and SO matrices satisfy the following properties:

(a) The union of all sets of i, i = 1, 2, ..., z rows of SI contains at least id+t ones.

(b) The union of all sets of j rows, j = 1, 2, ..., d, of SO overlaps each row of SI in at least j+t places, and the symmetric difference of each row of SI with the union of all sets of j rows, j = 1, 2, ..., d, of SO does not have more than d-j ones.

We now develop a few general procedures for synthesizing single-level separable and nonseparable SNPs, as well as algorithms for establishing the switch settings.

A3.3.3. SEPARABLE SNP SYNTHESIS

One procedure for synthesizing an SNP that is separable with respect to the input network is illustrated by means of the example of Figure A3.4, with parameters z=6, d=4, s=t=3. The general form of the input network for the case s=t is as follows. The first row contains switches in the first d+s positions. The second and all succeeding rows also contain d+s switches with an overlap of s switches with the preceding rows. Thus a given row is merely the preceding row shifted d places. It is seen that the total number of input switches is z(d+s).

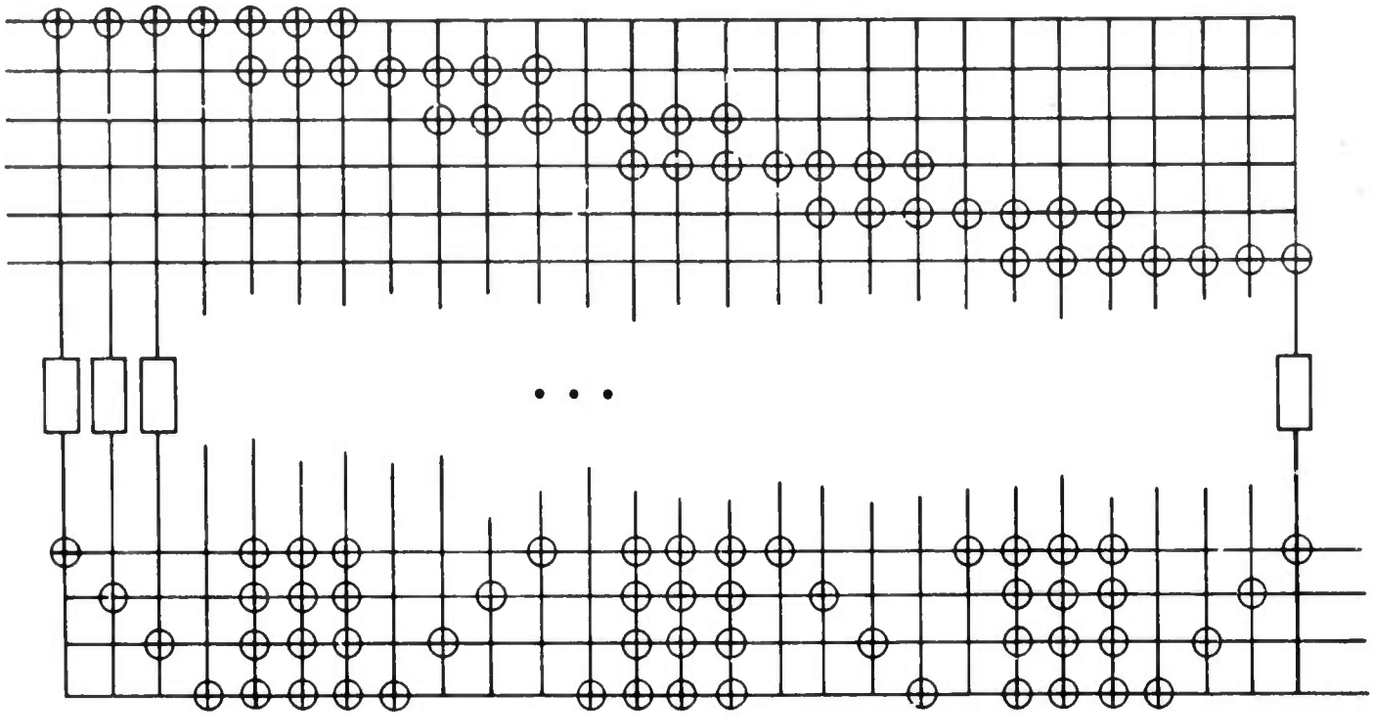


Fig. A3.4 An example of a separable SNP z = 6, d = 4, s = t = 3.

This input network bears some resemblance to Stiffler's "rippler" (see Stiffler 73). The rippler's function is to transfer data from (say) a d-byte register to an arithmetic unit containing d+R byte slices. The transfer is such that the order of the bytes is preserved while avoiding faulty byte slices. A reasonable form of the rippler network is the input network of Figure A3.4, where the number of switches per row is R+1, and the row overlap is R.

An extremely simple algorithm suffices for deciding which switches are to be set for the input network.

ALGORITHM 1. For each row in turn the d leftmost switches are considered. For each of these which corresponds to the position of a failed chip, this switch is skipped and the next to the right considered.

Now let us consider the output network as illustrated in Figure A3.4. The first d columns consist of a diagonal line of switches followed by a solid block of switches in columns d+1, d+2, ..., d+s. Thereafter the network consists of alternating diagonal lines of switches and "inverted" diagonals, with solid blocks of switches superposed on top of s consecutive columns every 2d columns. (The alternation of identity arrangements with the inverted identity arrangements provides a nearly balanced load on each row.) The number of switches in each row of the output network is bounded from above by $z + s\lceil z/2d \rceil$, yielding a total number of switches $zs + 2zd + sd\lceil z/2d \rceil$, including the input network; "$\lceil x \rceil$" denotes the smallest integer containing x.

An algorithm for deciding which switches are to be set in response to a pattern set by the input network is quite simple.

ALGORITHM 2. Consider the first d columns activated by the setting up of the input network. Switches are to be set in the output network so as to connect each of these d columns to a unique output row. First set the switches in the identity section to handle any of the activated columns. Those rows not yet served will be handled by setting appropriate switches in the solid block section. Then the second group of d columns is handled, and so on, until all groups are accommodated.

A3.3.4. NONSEPARABLE SNP SYNTHESIS

The primary advantage of separable SNP's is the simplicity of algorithms for deciding switch settings. One would expect that a price for such simplicity would be an increase in the number of required switches, but

we have not yet found a nonseparable SNP that is more economical than the separable network construction of Figure A3.4. However, one disadvantage of the SNP of Figure A3.4 is the excessive switch loading on some of the columns of the output network. This is a particularly severe problem if the switches are part of the memory chips. We have attempted to find separable SNPs wherein all columns in the output network contain an equal number of switches. Such networks can be found, but they are costly. This has led us to pursue the synthesis of nonseparable SNPs.

The nonseparable SNP displayed in Figure A3.5, for the parameters z=6, d=4, s=6, t=5, alleviates this difficulty by providing a nearly constant loading on all columns of the input and output networks. In this structure there is effectively one spare chip per control line (s=z). The guaranteed correction capability is t=5, independent of the other parameters. Each row of the input network contains 3(d+1) switches with an overlap of 2(d+1) switches between adjacent rows. The rows of the output network are simple cyclic shifts of a repetitive pattern, consisting of two switches followed by d-1 places with no switches.

The total switch count for this SNP is approximately 5z(d+1), or about twice that of the separable SNP of Figure A3.4 with t=5. The structure of Figure A3.5 can be generalized to one containing t(zd+s) switches, which tolerates all patterns of t or fewer chip failures.

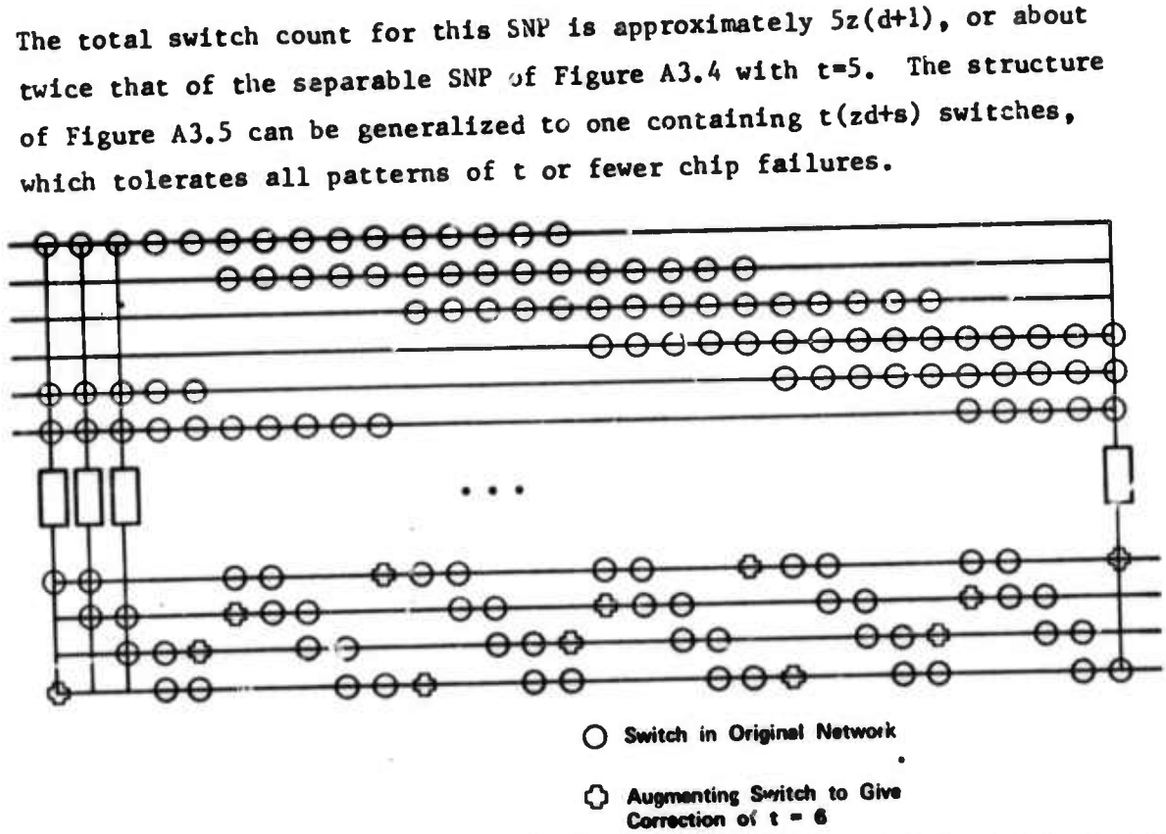


Fig. A3.5 An example of nonseparable SNP for z = 6, d = 4, s = 6, t = 5(6).

It is possible to increase the correction capability of this nonseparable SNP to t=6 by augmenting the output network with the extra switches indicated by ⬡ in Figure A3.5. This augmentation places an extra switch in all columns of the output network that previously contained only one switch. It is observed from Theorem 1 that each row of the output network must overlap each row of the input network in 7 places for a fault correction capability of t=6. Since the switches of each input row span three groups of output switches and since each such output switch group contains two columns of one switch, the augmentation technique yields the overlap of 7 only if $d \leq 6$.

This latter SNP is of interest from two viewpoints. First, the input and output switch loading is constant on all memory chips, i.e., the SNP is CHIP REGULAR. This regularity (or near regularity in the case of the nonaugmented version) permits the simple embedding of the input and output networks within the memory chips. Second, although there are some patterns of t+1, t+2, ... chip failures that are not correctable, the number of such offensive patterns for large values of z is small. In Sections A.3.3.6 and A.3.3.7 we discuss a realization with embedded switches and an analysis of the correction capability of the SNP beyond the guaranteed limit.

A3.3.5. MULTI-LEVEL NETWORKS

We thus see that there are SNPs that handle all combinations of t chip failures at a cost of approximately kzdt switches, where k is a constant between 0.5 and 1. This is certainly a tolerable cost for a relatively small number of chip failures, e.g., up to 8. However, if a large memory employing such reconfiguration techniques is to function unattended for a mission of a year or more, it might be necessary to handle 20 or more chip failures. In this case the switch cost can become a significant fraction of the total memory cost. As discussed below, the switch cost can in this case be reduced by replacing a single-level network by a multi-level network. The discussion below is brief, since a previous paper (Goldberg et al. 68) pursues the multi-level case in great detail -- although for a different application.

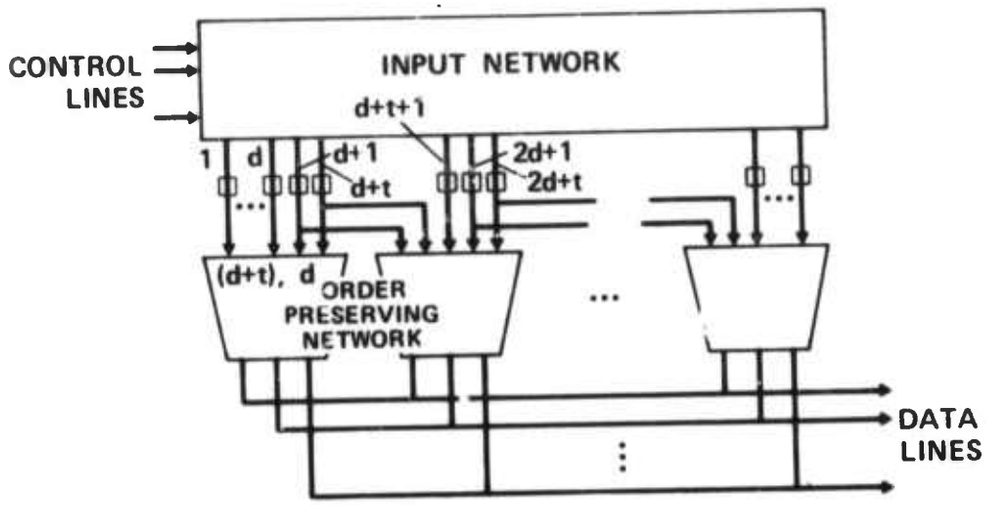


Fig. A3.6 A multi-level SNP.

Figure A3.6 illustrates the basic form of the SNP. The input network illustrated in Figure A3.4 requires only z(d+t) switches, a cost that is not excessive for all reasonable values of t. Thus the SNP of Figure A3.6 is assumed to have this same input network. However, the costly output network of Figure A3.4 can be avoided. Recall that the setting of the switches in the first row of the input network activates d columns among the first d+t columns. It is the role of the output network switches, corresponding to this set of d+t columns, to funnel the activated columns into the set of d output lines. Similarly, the second row of the input network activates d columns in the set d+1, d+2, ..., 2d+t, and so on for the remaining z-2 input rows. Hence, the output network function can be realized by a set of z order-preserving (OP) networks, each of which performs the funneling operation as described above. (Actually, for this memory application, the networks need not be order preserving, since the order of memory chips within a block of memory is not critical. However, if we require an efficient network, we have always been able to find an OP network as efficient as a comparable non-OP network.) The first OP network has as

input columns 1, 2,..., d+t, the second d+1, d+2, ..., 2d+t, the third 2d+1, 2d+2, ..., 3d+t, and so on. Each OP network yields d bytes. The ith bytes from each of these z networks are ORed together bytewise (e.g., by wired ORs) to form d bytes at the output.

In Goldberg et al. (68), a procedure is given for synthesizing such an OP network as an interconnection of two-input, two-output, two-state primitive cells as shown in Figure A3.7. Depending on the state of the cell, the inputs are interchanged or merely directed through the cell. We have described a recursive procedure for developing the network, as illustrated in Figure A.3.7. At the input, $\lceil (d+t-2)/2 \rceil$ cells and at the output $\lceil (d-2)/2 \rceil$ cells flank two smaller networks. The upper network is an OP network of $\lceil (d+t)/2 \rceil$ inputs and $\lceil d/2 \rceil$ outputs, while the lower is an OP network of $\lfloor (d+t)/2 \rfloor$ inputs and $\lfloor d/2 \rfloor$ outputs, where "$\lfloor x \rfloor$" is the largest integer contained in x. Each of these networks is replaced by a similar three-layer construction, and so on. Eventually, there is a degenerate requirement for an OP network of p inputs and 1 output. Such a network is easily realized as a simple linear array of p-1 cells.

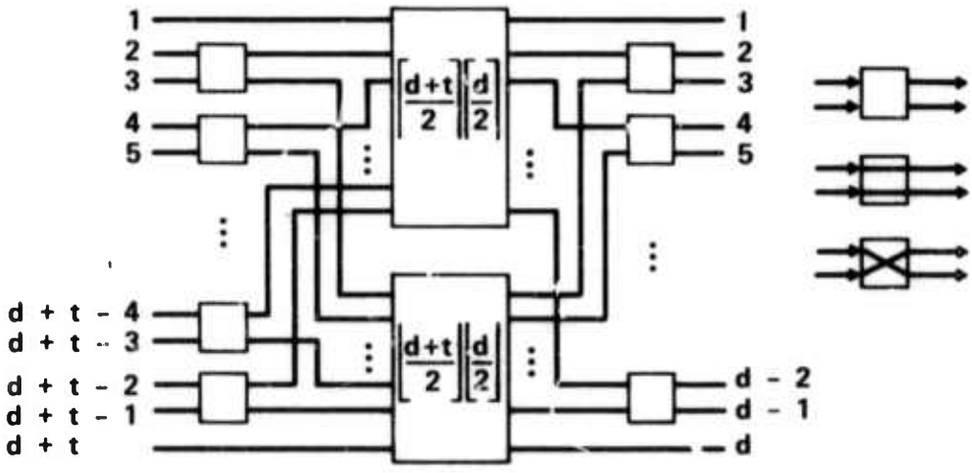


Fig. A3.7 Decomposition of the order preserving network.

The number of cells required for this OP network is $C(d+t)\log_2 t$, where C is approximately one-half. Thus, the number of cells in the output network (which still dominates the input network) is $Cz(d+t)\log_2 t$, which represents a saving of $t/\log_2 t$, compared with the single-level cross-bar realization. For $t > 8$, the multilevel version becomes more economical. Techniques for setting up the OP network and an approach to incorporating fault tolerance within it are discussed in Goldberg et al. (68).

In conclusion we have determined that the switch cost for reconfiguring the chips of a memory is small when compared with the total memory cost. In addition, we have shown that the algorithms for deciding which switches are to be set can be simple in certain cases.

A3.3.6. A NONSEPARABLE NETWORK WITH EMBEDDED SWITCHES

As mentioned previously the switches in the nonseparable SNP of Figure A3.5 can be embedded within the memory chips. In the augmented version of Figure A3.5, a given chip can, by virtue of the input switching network, receive an activation signal from one of three control lines, or be disconnected from all control lines. Similarly, by virtue of the output switching network, the chip can be switched onto one of two data lines, or be disconnected from all data lines. The process of embedding the switching within the chips can be seen by reference to Figure A3.8. Each chip has as inputs three control lines, and as outputs two data lines. An activation select switch makes the connection to one of three control lines, or to a fourth vacuous input. Similarly a data-line select switch makes the connection to one of two data lines or to the vacuous output.

Figure A3.9 shows the connections of the array of chips to the control and data lines for the same parameters as the SNP of Figure A3.5, namely, z=6, d=4, s=6, t=5 or 6. When the dotted line connections to the data lines are present, t=6 faults can be handled; otherwise, t=5.

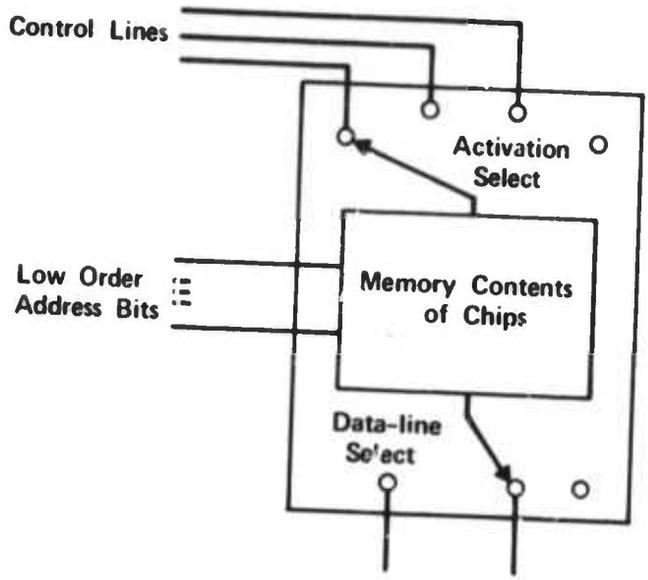


Fig. A3.8 Memory chip with components of input and output switches.

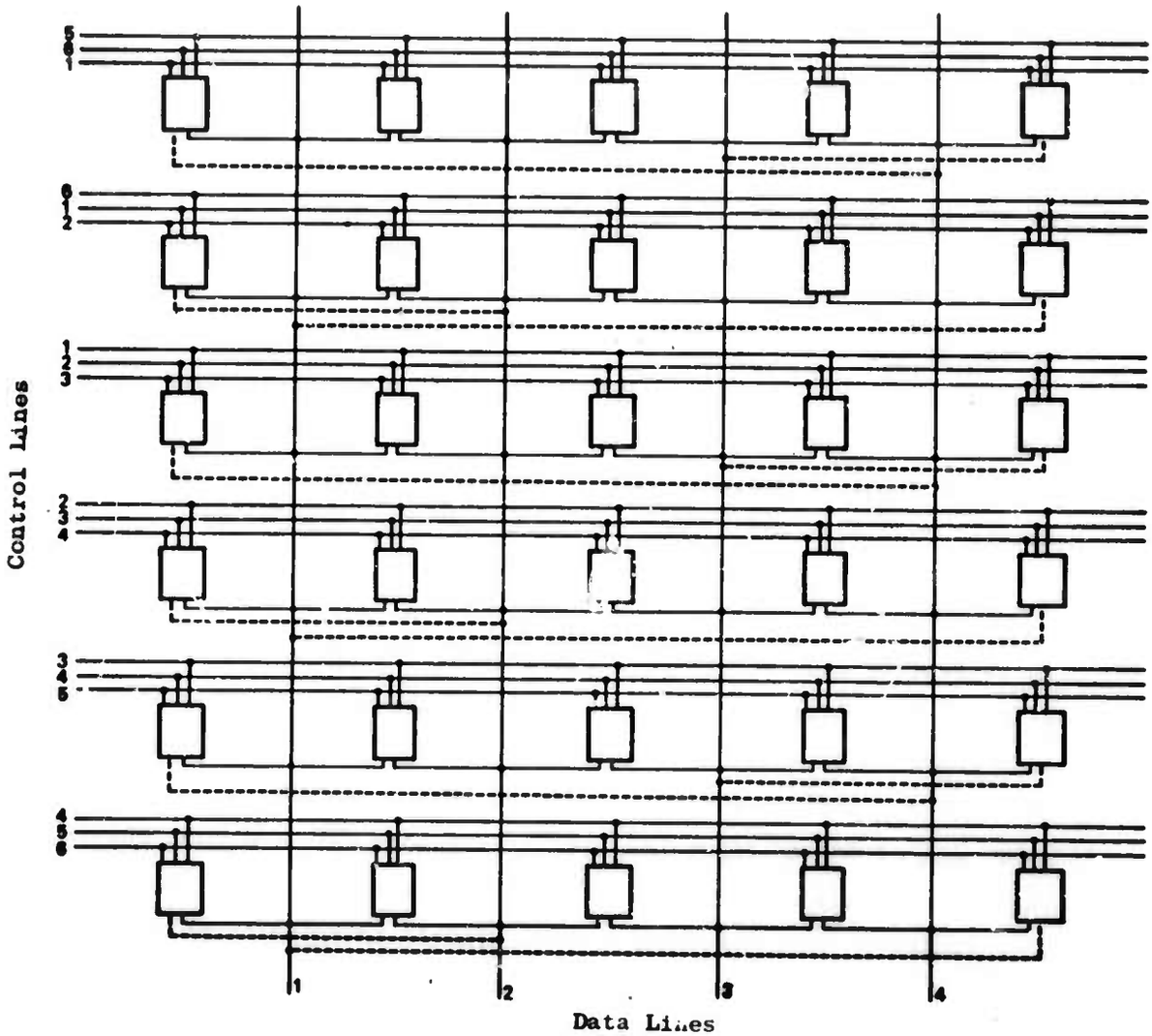


Fig. A3.9 Organization of nonseparable SNP with embedded switches

It is convenient to view the last column of chips as spares -- i.e., with all chips operative, this last column of chips remains disconnected. As failures occur, the spare chips are brought into service. We have developed an algorithm that determines the appropriate switch settings for any correctable fault pattern. The algorithm is more complicated than the algorithm for the separable case, and may require a substantial reorganization of the memory blocks subsequent to a failure, including operative blocks. The span of the reorganization can be shown (Goldberg et al. 68) to be related to the clustering of the chip failures in the array. That is, if the failures are spread out over the array, relatively little reorganization is required.

A3.3.7. ANALYSIS OF CORRECTION CAPABILITY IN REGULAR SNPs

The organization of the type depicted in Figures A3.5 and A3.9 exhibits more spares than the guaranteed fault-correction capability. However, in these organizations a large fraction of the fault patterns containing f failures, $t+1 \leq f \leq s$ are indeed correctable. In this section we present some approximate upper and lower bounds on the fraction of such faults that are correctable for the case of what we define below to be I/O regular SNPs. The derivation of these bounds is given below.

We define the function c(f) to be the fraction of patterns containing f faulty chips that cannot be accommodated by reconfiguration. In a memory organization with z rows each of d bytes, there are p=zd unique paths that must be established between the control lines and the data lines. The input and output switching networks must be set so that each path contains a (unique) nonfaulty chip.

In estimating c(f) we define a ROUTE to be the set of paths between a given control line and a given data line, and make the assumption that each route to be served contains e paths, and that e is a constant for each route. This is the definition of an I/O REGULAR SNP. The nonaugmented SNP of Figure A3.5 is I/O regular with e=6, but not chip regular. The opposite is true for the augmented version. (In particular some routes in the augmented version contain seven paths while others

contain eight.) Thus, the bounds derived below are only exact for the nonaugmented SNP. However, they represent lower bounds for the augmented case, provided the lower applicable value of e is used.

Clearly, we have the following special cases: $c(f) = 0$ for $f < e$, for clearly no route is deprived of all of its paths. On the other hand, if we consider the case where all spares are used, then $c(f)=1$ for $f > s$. The development of estimates (or bounds) for $c(f)$ reduces to the cases between these two extremes.

A particular fault pattern of f faulty modules will not be tolerated if and only if, for all $i \leq f$, it contains a sub-pattern such that all but (i-1) or less modules included in i routes are in the sub-pattern.

If we denote by c_i, the probability of the i^{th} term above, then

$$1 - c(f) = (1 - c_1)(1 - c_2) \ldots (1 - c_i) \ldots$$

For small values of c_i, a sufficiently close approximation is

$$c(f) \approx c_1 + c_2 + \ldots \tag{A13}$$

We introduce the concept of 'overlap' λ_{ij} defined by

λ_{ij} = number of modules that serve routes i and j in common, (A14)

and also

$$\lambda = \underset{i \neq j}{\text{MAX}} (\lambda_{ij}) \tag{A15}$$

The value of c_1 can be computed for regular structures (i.e., those for which e is a constant for all routes).

Given a pattern of f faults the number of sub-patterns of size e is $\binom{f}{e}$.

There exist just p patterns of e faults that will not be tolerated out of a total number of patterns of e faults of $\binom{m}{e}$; therefore:

$$c_1 \approx \frac{p f!}{(f-e)!\, m^e} \tag{A16}$$

For $m \gg e$, we have the approximation

$$c_1 = 1 - \left(1 - \frac{p}{\binom{m}{e}}\right)^{\binom{f}{e}} \qquad (A17)$$

Since c_2 , c_3 , ... are non-negative, (A17) is an approximate lower bound of c(f).

In obtaining an expression for c_2, we need to consider the size of a pattern that will not be tolerated because only one module remains of those that are included in two routes. Given two routes i and j, the minimum pattern to disable one of them because of commonality of modules to them is L_{ij} , where

$$L_{ij} = 2e - \lambda_{ij} - 1 \qquad (A18)$$

For the case where no overlap exists for routes i and j, i.e. $\lambda_{ij} = 0$, the disabling is of the type considered under the derivation of c above. We define the parameter L by

$$L = \underset{i \neq j}{\text{MIN}} (L_{ij}) = 2e - \lambda - 1 \ . \qquad (A19)$$

Given a pattern of f faults, there exist $\binom{f}{L_{ij}}$ sub-patterns of size L_{ij}. We consider each pair of routes i and j. For each such pair the fault pattern will be tolerated if and only if it does not contain a sub-pattern of size L_{ij} included in the set of modules of number $L_{ij}+1$ that serve the two routes i and j. For these $(L_{ij}+1)$ modules, there are $(L_{ij}+1)$ ways of selecting a fatal sub-pattern. Within the whole structure there are $\binom{m}{L_{ij}}$ sub patterns of size L_{ij} , and $\binom{f}{L_{ij}}$ are included in the fault pattern being considered. The pair of routes will survive with a probability Q_{ij} , where

$$Q_{ij} \approx 1 - \frac{(L_{ij}+1)\binom{f}{L_{ij}}}{\binom{m}{L_{ij}}} \qquad (A20)$$

the approximation being valid if $L \ll n$. The probability Q of all pairs surviving is given by

$$Q = \frac{1}{2}\prod_i \prod_{j\neq i} Q_{ij} = 1-c_2 \tag{A21}$$

whence

$$c_2 \approx \frac{1}{2} \sum_i \sum_{j\neq i} \frac{(L_{ij}+1)\ f!\ (m-L_{ij})!}{(f-L_{ij})!\,m!} \tag{A22}$$

Each term in the double series can be expanded in the form

$$(L_{ij}+1)\left(\frac{f}{m}\right)\left(\frac{f-1}{m}\right) \cdots \left(\frac{f-L_{ij}+1}{m-L_{ij}+1}\right) \tag{A23}$$

As $L \ll L_{ij}$, the replacement of L_{ij} by L in (A23) will result in fewer terms in the product. We can therefore derive an upper bound for c_2 as

$$c_2 \leq \frac{1}{2} \sum_i \sum_{j\neq i} \frac{(L+1)\ f!\ (m-L)!}{(f-L)!\,m!} = \hat{c}_2 \tag{A24}$$

which for $L \ll n$ yields the approximation

$$\hat{c}_2 \approx \frac{p(p-1)(L+1)\ f!(m-L)!}{2(f-L)!\,m!} \tag{A25}$$

Consider now the expression for c(f) the probability of non-coverage, i.e. from (A13)

$$c(f) = c_1 + c_2 + \cdots \tag{A26}$$

On intuitive grounds, we say that this series is strongly converging, for if it were not so, the implication would be that a fault pattern would be more probable to be not covered because of interaction among (i+1) routes than between i points. For values of i small compared to m, this implies that in going from i to (i+1) there is a greater probability of the new fault being strongly connected than being disjoint, which for small i is absurd. We therefore consider only the first two terms of the series to obtain

$$c(f) \approx c_1 + c_2 \tag{A27}$$

or

$$c_1 \leq c(f) < c_1 + \hat{c}_2 \tag{A28}$$

Note: in computing values of c_1 and c_2, the c_2 is, for reasonable

cases, significantly smaller than c_1, lending credence to the intuitive argument above. The bounds on c(f) are therefore:

$$\frac{pf!}{(f-e)!m^e} \leq c(f) \leq \frac{pf!}{(f-e)!m^e} + \frac{p^2 (L+1)f!}{2\, m^L (f-L)!} \qquad \text{(A29)}$$

A3.3.8. REGULAR SEPARABLE SWITCHING NETWORKS

We consider the design of input (SI) and output (SO) switching networks which are separable and are also uniform in that the fan-in and/or fan-out of each unit of each part of the system (decoder, chip, etc.) is the same.

Define:

- b = number of inputs to SI
- d = number of outputs from SO
- s = number of spare chips
- t = number of faulty chips to be tolerated = s
- m_i = number of cells in each row of SI
- m_o = number of cells in each row of SO
- k_i = number of cells in each column of SI
- k_o = number of cells in each column of SO
- m = number of chips total = bd+s

Separable networks have the desirable property that a simple algorithm is known for setting the switches in the presence of arbitrary fault patterns. Most separable networks known to date have the disadvantage that the loading on the parts of the system is nonuniform. We develop a set of necessary conditions for a network to be regular separable. Using these conditions a number of potentially regular separable networks (RSN) have been found, some of which are indeed RSN. No cases have been found of a network satisfying all the conditions and not being RSN. We conjecture that all the cases are RSN.

CONDITION 1--REGULARITY OF SI

The total number of cells in SI is bm_i . The total number of modules is m=bd+s. Clearly.

$$\frac{bm_i}{bd+s} = k_i \quad \text{(an integer)} \tag{A30}$$

By Theorem 1 of Section 4.2, it follows that $m_i = d+s$:

$$\frac{b(d+s)}{(bd+s)} = k_i \tag{A31}$$

or

$$s = \frac{(k_i-1)bd}{(b-k_i)} \tag{A32}$$

CONDITION 2--REGULARITY OF SO

The total number of cells in SO is $m_o d$. Clearly,

$$m_o d/(bd+s) = k_o \text{ where } 1 < k_o < d \text{ and } k_o \text{ integer} \tag{A33}$$

CONDITION 3--SEPARABILITY

We restate Theorem 2 of Section 4.2 on necessary and sufficient conditions on SO.

"Assume a valid SI, then the combination of SI and SO is separable if and only if the union of every set of j rows of SO (j=1...d) overlaps each row of SI in at least s+j places."

Consider any row of SO and apply a test for the case j=1. The overlap with the first row of SI must be at least s+1. We must therefore allocate at least s+1 cells to those columns where the first row of SI has cells. Call this allocation A_1. In making the allocation there are $k_i A_1$ cells in the allocated columns of SI. Consider another row of SI, choosing that one that has a minimum number of cells in the columns already allocated. There are $(k_i-1)A_1$ available cells in the allocated

columns to be shared over the remaining b-1 rows. Therefore, there exists at least one row of I which contains only $\lfloor (k_i-1)A_1/(b-1) \rfloor$ cells in the allocated columns. Choose this row. We must allocate more cells of the row of SO to this row of SI. Specifically we must allocate at least $(s+1)-\lfloor (k_i-1)A_1/(b-1) \rfloor$. Specifically we must allocate at two rows is therefore

$$A_2 = A_1+(s+1)-\lfloor (k_i-1)A_1/(b-1) \rfloor \qquad \text{(A34)}$$

Using the above reasoning to successive rows of SI we can develop the general form

$$A_1 = s+1$$

$$A_\ell = A_{(\ell-1)}+(s+1)-\lfloor (A_{(\ell-1)}k - (s+1)(\ell-1))/(b-\ell+1) \rfloor \quad \ell = 2 \ldots b \qquad \text{(A35)}$$

The necessary condition on m_o becomes $m_o \geq a_6$. Note that Condition 3 is necessary, but to prove sufficiency using Theorem 2 of Section 4.2, it is required to consider every set of j rows of SO. It is, however, conjectured that for regular networks, Condition 3 may be sufficient. No cases that satisfy Condition 3 that are not regular separable have been found.

To illustrate the test consider the case b=6, d=4, s=6, k=2, m=10. Then $A_1 = 7$, $A_2 = 13$, $A_3 = 17$, $A_4 = 20$, $A_5 = 21$, $A_6 = 21$, whence $m_o \geq 21$.

The scheme used to find RSNs was programmed and the results of a small run are shown in Tables A1 and A2. Note that the solution m = bd+s which is a totally full SO is trivial and is not shown.

We conclude that regular Separable Network exist, but such networks contain a very high proportion of cells in the switch, leading to high fan-out and fan-in. However such networks can be designed to enable reconfiguration in the presence of a large number of faults.

Table A1 Potential RSNs for $k_i = 2$

b \ d	4	5	6	7	8	9	10	11	12
4	8/18	10/24	12/30	14/30, 36	16/36, 42	18/42, 48	20/42	22/48, 54,60	24/54, 60,66
5						15/40			
6			9/30		12/45		15/60		18/60, 75
7									
8						12/56			16/84
9									
10									15/90
11									
12									

Note: The value of s and the (possible) multiple values of m_o are indicated in the form $s/m_o, m_o \ldots$, for example the entry 18/42, 48 indicates the case for s = 48 and m_o = 42 or 48. A maximum of three values of m_o are shown.

Table A2 Potential RSNs for $k_i = 3$

d

b	4	5	6	7	8	9	10	11	12
4							80/108	88/120	96/132
5				35/60	40/70	45/80	50/90	55/100	60/110
6			24/50	28/60	32/70	36/80	40/90	44/90, 100	48/100, 110
7							35/84		42/105
8									
9		15/48	18/60	21/72	24/84	27/84, 96	30/96, 108	33/108, 120	36/120, 132
10									
11									
12									32/132

APPENDIX 4

ERROR CORRECTION IN BYTE-ORGANIZED ARITHMETIC PROCESSORS*

Peter G. Neumann
Computer Science Group
Stanford Research Institute
Menlo Park, CA 94025

and

T. R. N. Rao
Electrical Engineering Dept.
University of Maryland
College Park, MD 20742

ABSTRACT

This paper considers codes with radix $r \geq 2$ which are capable of correcting arbitrary arithmetic errors in any radix r digit. If each radix r digit represents a byte of b binary digits (e.g., $r = 2^b$), these codes correct any combination of errors occurring in the b binary digits of any single byte. A theoretical basis for these codes is presented, along with practical considerations regarding their applicability.

I INTRODUCTION

This paper is concerned with error detection, error correction and error location for multiple errors within a particular byte of an arithmetic unit, and is motivated by several observations. First, it is possible to obtain byte error-correcting arithmetic codes with low code redundancy. Second, it is possible to provide high system availability and relatively maintenance-free operation through autonomous replacement with spares. For certain applications it is desirable to replace not an entire processor or arithmetic unit, but rather one of several identical sub-units. Thus byte-slicing is attractive. Third, byte-slicing is also naturally allied with fast-carry logic, e.g., carry look-ahead over bytes (and even within bytes). Fourth, LSI technology is suitable for realization of a byte of logic on a chip. Fifth, LSI technologies often give rise to multiple errors on a chip resulting from a simple fault. Thus higher radix (byte) arithmetic coding may be highly effective: with chips corresponding to bytes, multiple errors in arithmetic within a byte may then be economically corrected. Besides, single-bit error-correcting codes are inadequate for the multiple errors which may arise from fast-carry logic. Location of the faulty byte-slice and autonomous replacement with spares is also facilitated by the byte coding.

In this paper previous results of Peterson and of Rao and Trehan for perfect single-error-correcting arithmetic codes are generalized to higher-radix number systems. A single arithmetic error in a radix r representation is of the form $\pm ar^j$, $0 < a < r$. It is shown here that all such errors are correctable by an AN code with generator A of the form $(r-1)p$, where p is a prime greater than r satisfying certain specified conditions. These results also apply directly to corresponding systematic codes (e.g., bi-residue codes with residues r-1 and p, and gAN codes). Further results are also given for other interesting (but non-perfect) codes.

The results of this paper are potentially suitable for use in a byte-organized processor, e.g., using one chip for each b-bit byte (representing a radix r digit) of the processor, where $r = 2^b$, r = 10, etc. Thus it is possible to correct any combination of bit errors resulting from errors in any single byte position, that is, any arbitrary single-digit arithmetic error in the higher radix r. As a consequence, certain known bit correcting codes are seen to be byte correcting as well. Examples are included here, along with a discussion of the applicability of such codes in fault-tolerant computing systems. To determine the set of all possible errors capable of arising from various faults, a careful and thorough analysis is required, such as the one conducted by Langdon and Tang [12] for adders employing carry look-ahead between and within bytes. Their analysis establishes that the errors in carry look-ahead adders resulting from single faults are frequently not of the form $\pm 2^j$. Therefore the binary single-error-correcting codes are not effective in such cases, especially in byte-per-chip realizations. Here we assume that the byte adders can be designed in such a way that the carry-out (look-ahead) logic circuit is independent of the rest of the logic (namely, the internal carry generation, sum-byte logic, etc.). Consequently, we allow any error combination in the sum byte or in the carry-out but not in both (unless that combination is equivalent to an error in one or in the other). Specifically, the byte-correcting codes discussed here are capable of correcting any additive error involving a single digit (byte) of radix r, of the form αr^j, where α is a positive or negative additive error of magnitude $a = |\alpha|$, $0 < a < r$, and where j is the position of the radix r byte processor in error, $0 \leq j < \hat{n}$. Such errors are characterized as single arithmetic errors in radix r by Peterson [17] and have arithmetic (Peterson) weight one. More precisely, in adders using radix-complement (or diminished-radix-complement) arithmetic, a single byte (arithmetic) error E is defined as an error of modular weight one [21], and is given by

$$E = \alpha r^j \text{ or } m - \alpha r^j,$$

where $0 < |\alpha| < r$, $0 \leq j < \hat{n}$, and $m = r^{\hat{n}}$ ($m = r^{\hat{n}} - 1$ for the diminished radix complement case).

II BYTE-ERROR DETECTION

Error detection techniques are well known using "AN" codes (which are nonsystematic) [4,5,17] and "$(N, |N|_A)$" residue codes (which are systematic and separate) [1, 4, 18,19]. Single-byte error detection arises whenever the base A is an integer greater than r that is relatively prime to r. Two suitable choices are $r + 1$ and $r^2 - 1$. When $r = 2^b$, the check base $r - 1$ is also interesting, particularly for the simplicity of its implementation; however, in this case not all single byte errors are detected (e.g., an error changing 0 to $r - 1$ leaves the residue unchanged), although the most probable errors [1] and a very high percentage of all single byte errors are detected. Such a residue code is used (with b = 4) in the JPL STAR computer [3].

III NEAR-PERFECT BYTE-CORRECTING CODES

Single-error correction in binary adders is attainable with the nonsystematic AN codes first studied by Brown [5] and Peterson [17], and with the systematic multi-residue codes studied by Avizienis [1, 2, 4], Rao [18 19, 21] and Garcia. Systematic rearrangements of the AN codes, namely, the systematic gAN codes, have been discussed by Garner [9] and by Rao [20]. Extension of the results of Brown and Peterson to higher radices have been discussed previously by Rao and Trehan [22], primarily for r = 3. Here we first characterize those optimal AN codes in radix r which are capable of correcting arbitrary arithmetic errors in a single (radix r) byte. These byte-correcting codes are obtained by choosing A of the form (r-1)p, where p is a prime greater than r satisfying certain specified conditions. (Theorems of Peterson and of Rao and Trehan follow as special cases for r = 2.) Further theorems are developed which aid in deriving suboptimal codes, and examples are cited. These results are immediately applicable to corresponding systematic codes with the same r and p: bi-residue codes $(N, |N|_{r-1}, |N|_p)$ in radix r with residues r - 1 and p, and gAN codes with A = (r-1)p, both of which are therefore also byte correcting. Interesting byte-correcting codes also exist for some nonprimes p.

The reader is assumed to be exposed to the concepts of arithmetic weight, arithmetic distance, and linear congruences used here; he may wish to refer to Peterson [17] or to Massey and Garcia [14] for background. Throughout Sections III and IV, p denotes a prime greater than r. The following are observed throughout this paper: $G_r(p)$ denotes the cyclic (multiplicative) subgroup $\{r^j \pmod p\}$, and $e_r(p)$ denotes its order; $e_r(p)$ is also called the order or exponent of r in the field GF(p); "a" is a nonzero radix r digit, $0 < a < r - 1$, i.e., an element of the field; a^{-1} is its multiplicative inverse, with $aa^{-1} \equiv 1 \pmod p$. With A = (r-1)p, $M_r(A,3)$ is the maximum number of code words in the radix r byte-correcting AN code (with arithmetic distance 3). The error syndrome of a given presumed word in an AN code is the modulo A residue of that word, e.g., 0 if it is a correct code word AN, since every code word has residue zero. Thus an error αr^j, $0 < |\alpha| < r$, has the syndrome $\alpha r^j \pmod A$. With this background the following theorem is the basic theorem of this paper.

Theorem 1: For any prime $p > r$, given that p - 1 does not exist in $G_r(p)$ and that the condition

$$(a-r+1)a^{-1} \notin G_r(p) \text{ for all } a,\ 0 < a < r - 1 \qquad (1)$$

is satisfied, then

$$M_r(A,3) = \frac{r^{e_r(p)} - 1}{A} . \qquad (2)$$

Proof is found in the Appendix, along with proofs of other theorems. (As an example, the reader might try r = 8, p = 19, a = 2.) Next we consider the special case when -r (i.e., p-r) is primitive [i.e., $(-r)^i$ (mod p) generates all elements from 1 to p-1 for i from 0 to p-2], but when r is not primitive in GF(p). We know from number theory (e.g., [19]) that in this case $e_r(p) = (p-1)/2$, while $(-r)^{(p-1)/2} \equiv -1 \pmod p$ and $r^{(p-1)/2} \equiv 1 \pmod p$. Therefore (because of the nonprimitivity of r) -1 does not exist in $G_r(p)$, satisfying the first part of the hypothesis of Theorem 1. This provides us with the following useful result.

Theorem 2: Given that -r, but not r, is primitive in GF(p) and that condition (1) is satisfied, then

$$M_r(A,3) = \frac{r^{(p-1)/2} - 1}{A}, \quad A = (r-1)p. \qquad (3)$$

Theorems of Peterson [17] and of Rao and Trehan [22] follow by setting r = 2 and r = 3, respectively in Theorem 2, since condition (1) is valid.

Corollary 3 (Peterson): If -2 but not 2 is primitive in GF(p), then

$$M_2(A,3) = \frac{2^{(p-1)/2} - 1}{A} , \quad A = p . \qquad (4)$$

Corollary 4 (Rao and Trehan): If -3 but not 3 is primitive in GF(p), then

$$M_3(A,3) = \frac{3^{(p-1)/2} - 1}{A} , \quad A = 2p . \qquad (5)$$

The sequence of expressions (4) and (5) extends readily to r = 4.

Theorem 5: If -4 but not 4 is primitive in GF(p), then

$$M_4(A,3) = \frac{4^{(p-1)/2} - 1}{A}, \quad A = 3p . \qquad (6)$$

For r > 4, however, the simplicity of (4), (5) and (6) no longer exists. Condition (1) is no longer generally satisfiable, and we must resort to Theorem 2. (When condition (1) is not satisfied, Theorem 7 below is useful.)

Theorem 2 is thus a generalized form of the Peterson Theorem whenever -r but not +r is primitive. Its converse is also true. The full theorems of Peterson and of Rao and Trehan also cover the case of +r primitive for r = 2 and 3, respectively, for which cases p-1 is in $G_r(p)$:

$$M_r(A,3) = \frac{r^{(p-1)/2} + 1}{A} , \quad r = 2, 3, \qquad (7)$$

if and only if +r is primitive in GF(p). Unfortunately, (7) does not hold for any r > 3, since r-1 cannot divide $r^{(p-1)/2} + 1$ for any p. A counterpart of Theorem 1 exists in this case, however, as follows.

Theorem 6: Given that p-1 exists in $G_r(p)$, and that condition (1) is satisfied, then

$$M_r(A,3) = \frac{r^{e_r(p)/2} + 1}{p} \quad \text{for r even,} \qquad (8)$$

$$= \frac{r^{e_r(p)/2} + 1}{2p} \quad \text{for r odd.} \qquad (9)$$

Theorem 2 specifies the existence (or nonexistence) of near-perfect codes in which all possible nonzero syndromes (omitting r-2 multiples of p) are used to correct the possible byte errors in each of the $\hat{n} = (p-1)/2$ bytes of the resulting radix r AN code. The codes covered by (7) and by Theorem 2 (and its derivatives (4)-(6)) are the only near-perfect byte-correcting AN codes. (Those for r = 2 are perfect.) The hypotheses for Corollary 3 and Theorem 5 are true precisely when p = 8i-1 and 4i-1 are both primes (cf. [23], Theorems 38 and 39). Such codes therefore exist for r = 4 (as well as r = 2) when p = 7, 23, 47, 71, 79,... . As further examples, the shortest nontrivial near-perfect codes for r = 5, 6, 7, 8, 9, and 10 have p = 11, 19, 31, 71, 59, and 31, respectively. The shortest nontrivial near-perfect code for r = 16 has p = 503. (Note that p must be at least 2r-1 for a code to be perfect.)

IV NONPERFECT BYTE-CORRECTING CODES WITH PRIME p

For completeness, the following theorem is included as an extension of Rao and Trehan's [22] Theorem 5. It is sometimes helpful in generating efficient codes, particularly for $r > 4$.

Theorem 7: When condition (1) does not apply, let j be the smallest possible positive integer such that $rj \equiv c \pmod p$, where $c \equiv (a-r+1)a^{-1} \pmod p$ for some a in $0 \le a \le r-1$. That is, $c \equiv -\hat{a}a^{-1}$, where $\hat{a} = r-1-a$. Then

$$M_r(A,3) = \frac{ar^j + \hat{a}}{A} \quad .$$

A useful class of nonperfect byte-correcting codes is available when p is a (Mersenne) prime of the form 2^d-1. Corresponding results for nonprimes of this form are given in the next section. Residues of this form are called "low-cost" by Avizienis [1] because of the relative simplicity of implementation.

Lemma 8: Given $r = 2^b$ and prime $p = 2^d - 1$, $p > r$, it follows that condition (1) is satisfied.

Theorem 9: For $r = 2^b$ and prime $p = 2^d - 1$, $p > r$,

$$M_r(A,3) = \frac{r^d - 1}{A} \quad . \tag{10}$$

Theorem 8 follows from Lemma 8 and Theorem 1. The resulting codes correspond to the single-bit error correcting biresidue codes [1, 19] with the given residues r-1 and p (prime), which are thus seen to be byte correcting as well as bit correcting.

V EXTENSION TO NONPRIMES p

The foregoing theorems all assume that p is a prime. However, byte-correcting codes in fact exist for many nonprimes p--although none is near-perfect. An example is $r = 8$, $p = 11 \cdot 13$, $AM_8(A,3) = 2^{60} - 1$, which arises from an extension of Theorem 1. For low-cost residues, Theorem 8 is generalized below (with an additional condition) to certain nonprimes $p = 2^d - 1$. For this case, nontrivial codes exist for every $d \ge 3$ other than 4 and 6, for at least some $r = 2^b \ge 4$.

Theorem 10: With $r = 2^b$ and $p = 2^d - 1$ ($d > b$, with p not necessarily prime), with gcd (r-i, p) = 1, and with $A = (r-1)p$, let f be the largest integer $1 \le f < d$ for which $2^f - 1$ (also not necessarily prime) is a divisor of p. Then

$$M_r(A,3) = \frac{r^d - 1}{A} \quad \text{iff } r \le \frac{p}{2^f - 1} \quad .$$

As a consequence of Theorem 10, some but not all single-bit error correcting biresidue codes with residues $r-1 = 2^b-1$ and $p = 2^d-1$ are in fact also byte correcting, above and beyond those covered by Theorem 9. For $p = 255$, for example, the code with $r = 8$ is byte correcting, while the codes with $r = 32$ and 128 are not (unless truncated to about half their length). For $p = 2047 = 23 \cdot 89$, the codes for all $r = 2^b$, $1 \le b < 11$, are byte correcting.

If p is generalized to

$$p = \prod_{i=1}^{j} (2^{d_i} - 1)^{t_i}, \quad t_i \ge 1 \ , \tag{11}$$

some additional simply implementable and more efficient byte-correcting codes arise, with each $d_i > b$, with pairwise gcd's all one among the d_i's and b, where each value of $2^{d_i} - 1$ satisfies $r \le (2^{d_i} - 1)/(2^{f_i} - 1)$, where $2^{f_i} - 1$ is the largest such divisor of $2^{d_i} - 1$, $1 \le f_i < d_i$. A simple example of such a byte-correcting code has $r-1 = 3$, $p = 49 = 7^2$ (a base 7 residue calculation!), with $AM_4(A,3) = 2^{42} - 1$. This code is close to near-perfect (cf. Theorem 42 of [23]). (It is related to the more redundant code with $p = 7 \cdot 127$.) Such codes include certain of the multi-residue codes [1,21], including not just those with prime residues $2^{d_i} - 1$ ($t_i = 1$), but also some with nonprimes. A simple example of the latter type has $r-1 = 7$, $p = 31 \cdot 255$, for which $AM_8(A,3) = 2^{120} - 1$. (Note that the code with $r-1 = 7$, $p = 15 \cdot 31$ has $AM_8(A,3) = 5 \cdot 2^{30} + 1$, although the triresidue code has $AM_2(A,3) = 2^{60} - 1$ for the same A.) Thus the greedy algorithm of simply trying multi-residue codes does chew off various Mersennary codes that are byte correcting.

VI SOME POTENTIALLY USEFUL EXAMPLES

Arithmetic coding is of interest for words of length up to about 64 (or possibly 128 for applications such as double precision and multiplication). Table I illustrates some byte-correcting codes for $r = 4$, 8 and 16, and for $r = 10$. Values of p, $AM_r(A,3)$, r, ρ_A and ρ_B are given in the table, with the following meaning. The AN codes for $A = (r-1)p$ can be used to encode up to $M_r(A,3)$ code words; ρ_A is the effective bit redundancy required by A. The given value of n is such that 2^n is the largest power of two contained in $AM_r(A,3) + 1$. Thus $n - \rho_A$ is the effective number of binary information digits in the AN code.

On the other hand, the bi-residue codes with residues r-1 and p can be used to encode up to $AM_r(A,3)$ code words; ρ_B is the bit redundancy required by these codes. The given value of n is thus the effective number of binary information digits in the bi-residue code. If syndromes are computed in bi-residue form in both cases, then corresponding byte errors have identical syndromes. (Note that when only one value of ρ_A and ρ_B is given, it is the value of both.) The results also apply directly to the systematic gAN codes [8, 20], providing a permuted subcode of the AN code with $2^{n-\rho_A}$ code words.

Near-perfect codes in the table (derived from Theorems 2 and 5) are indicated with asterisks. The remaining codes in the table are all derived from Theorem 7, with the exception of those with $r = 8$, $p = 17$, and with $r = 16$, $p = 31$ (which arise from Theorems 6 and 9, respectively), and that with $r = 16$, $p = 73$ (which arises from Theorem 1, but which is closely related to a Theorem 10 code with $p = 511$, $\rho = 13$). Table II summarizes a few selected codes with p given by (11.) Since near-perfect codes are seen to be fairly sparse for $r = 2^b \ge 8$ and reasonable n, the codes of the form of (11) are often competitive in terms of redundancy, besides having implementation advantages.

p	Values of n for		
	r = 4	r = 8	r = 16
511	18	--	36
2047	22	33	44
$2^{17}-1$	34	51	68
7^2;7·127	42	--	--
31·127	70	105	140

Table II. Examples of single-byte correcting arithmetic codes with simple syndrome generation.

n ≥	r = 4, b = 2				r = 8, b = 3				r = 16, b = 4				r = 10			
	p	AN_r	n	ρ_A,ρ_B	p	AN_r	n	ρ_A,ρ_B	p	AN_r	n	ρ_A,ρ_B	p	$AN_r(A,3)$	n	ρ_A,ρ_B
16	23•	$4^{11}-1$	22	7	17	$7\cdot8^{4}+1$	14	7,8					19	$2\cdot10^{4}+7$	14	8,9
					29	$5\cdot8^{7}+2$	23	8	31	$16^{5}-1$	20	9				
32					41	$5\cdot8^{8}+2$	26	9	53	$12\cdot16^{6}+3$	27	10				
	47•	$4^{23}-1$	46	8					73	$16^{9}-1$	36	11	31•	$10^{15}-1$	49	9
64	71•	$4^{35}-1$	70	8,9	53	$6\cdot8^{2C}+1$	62	9	101	$14\cdot16^{10}+1$	43	11				
	79•	$4^{39}-1$	78	8,9					139	$10\cdot16^{17}+5$	71	12	107	$7\cdot10^{19}+2$	65	11
					71•	$8^{35}-1$	105	9,10	263	$8\cdot16^{34}+7$	139	12,13				
									503•	$16^{251}-1$	1004	13				

Table I. Summary of byte-correcting arithmetic codes for various radices and small primes p.

VI ERROR LOCATION

In practice there may be no need for error correction (apart from real-time criticalities) if the faulty byte processor can be immediately located and replaced by a spare. Alternatively this byte processor could be removed, with computation continuing either with degraded precision or with multiple precision operations. (Instruction retry without replacement may of course be adequate if the fault is transtent or intermittent.) Thus the use of error-locating arithmetic codes which specify the byte processor in error might appear to be very desirable. Unfortunately (with exceptions noted below) almost all linear byte-error locating codes are also byte-error correcting. This follows from the linearity of the syndrome generation for errors within a byte position--which errors therefore have distinct syndromes. Of course, error-correcting codes may be used as error-locating codes. (Partial error location is discussed in [1,4].)

Error-locating codes that are not error correcting do in fact exist: outright duplication and comparison has this property since the lowest byte position exhibiting a discrepancy is the position in error--assuming that the arithmetic error was confined to a single byte. (Note that duplication of n-bit words can be considered as an AN code in which $A = 2^n+1$.)

VIII MULTIPLE BYTE-ERROR DETECTION AND CORRECTION

AN codes for r = 2 are known that are capable of detecting double errors while correcting single errors (distance 4, e.g., for A = 43), or of correcting double adjacent errors (e.g., for A = 41)--see [17]. Similar codes also exist for r > 2, along with corresponding multi-residue codes. As an example, consider r = 4, p = 109. Using the residues 3 and 109 over 2-bit bytes results in an AN code with single-byte error correction plus double-byte error detection with $M_4(327,4) = 9$. The corresponding bi-residue code has up to $AN_4 = 2943$ code words, or at least 11 bits of information with 9 bits of redundancy.

IX SOME IMPLEMENTATION CONSIDERATIONS

The codes discussed here offer considerable flexibility and effectiveness in use with byte-sliced arithmetic units [7,24], permitting replacement of faulty byte processors. The cost of reliably switching the spares does not seem to be excessive (e.g., [19]). However, a careful comparison remains to be made with alternative schemes involving replication, comparison, and diagnosis, under various system assumptions. In any event the total system effect must be considered in time and in equipment complexity. Results of Rao [19] for r = 2 seem to indicate that about a 100% increase in equipment (i.e., effectively equivalent to duplication of the arithmetic unit) suffices to provide byte-error correction.

Various arguments concerning the implementation of arithmetic codes are also relevant here (cf. [1,2,4,7, 18-22]. In general the effectiveness of arithmetic coding using arbitrary residues p other than of the form (11) rests heavily on the effectiveness of the residue calculations, possibly even using analog techniques [6]. The use of low-cost residues p of the form $2^d - 1$, or more generally of the generalized low-cost residues (11), simplifies syndrome calculation. Note however that various tricks may also be useful. The residue modulo 49 is not needed in the code with r=4 and p=49 unless an error has actually occurred; single-byte errors are completely and rapidly detectable by use of the residue 7 alone. For the code with r=16 and p=73, residues modulo 73 may be derived from residues modulo 511, since 511=7x73.

Byte-correcting arithmetic codes also provide byte-error correction when used in memory, e.g., in a byte-organized memory [10]. As seen below, some of these codes have redundancy very close to the best comparable byte-error correcting codes for memory [11] given by Hong and Patel. Such codes thus have potential for dual use both in memory (for error correction) and in arithmetic (for error detection at least, if not for error correction). Advantages of such dual use are similar to those in the JPL-STAR [3], which uses a (modified) residue 15 error-detecting code. See also [16].

For comparison purposes, the redundancies of byte-correcting codes of various types are summarized in Table III. Included are the byte-correcting Hong-Patel codes for memory, denoted by "M" in the table, and the following arithmetic codes: the AN and gAN codes (denoted by "A"), (multi-)residue codes with arbitrary residues ("R"), (multi-)residue codes with generalized "low-cost" residues of the form (11) (denoted by "G"), and those (multi-)residue codes with low-cost residues, of the form 2^d -1 (denoted by "L").

The near-perfect AN codes (when they exist) have redundancy

$$\log_2 [(r-1)p],$$

at most one bit more than that of the byte-correcting Hong-Patel codes for memory [11], which require a redundancy of at least the larger of 2b and

$$\log_2 [(r-1)(n/b) + 1] = \log_2 [(r-1)(p-1)/2 + 1].$$

(The Hong-Patel redundancy is actually equal to this latter number in many cases.)

Arithmetic codes need not be near-perfect to be close in redundancy to the Hong-Patel memory codes. Several examples of such potential dual-use arithmetic codes are worth noting. One case is that with r=4 and with k from 37 to 42. Here 7 bits of redundancy are required for byte correction in memory (M), while 8 bits suffice for several forms of arithmetic byte correction (A, R, G), e.g., the AN codes with A=3x71 and with A=3x79, and the generalized low-cost residue code with p=49. (Note that the Hamming code for single-bit error correction requires 6 bits of redundancy.) Other examples with this one-bit-extra property exist for b=3 with k from 45 to 62 (with 8 bits of redundancy for M, 9 bits for A and R); for b=6 with k up to 42 (with residues 63 and 127 giving ρ=13, instead of 12 for byte correction in memory alone); and for b=10 with k up to 110 (with residues 1023 and 2047 giving ρ=21, instead of 20). In many cases, however, the arithmetic code redundancy is significantly greater than the memory code redundancy. In such cases the arithmetic codes may not be suitable for dual use, although they may still be applicable for arithmetic alone.

In passing, it is worth noting two oddities for p=31 (a low-cost residue for $r=2^b$), namely the near-perfect codes with radices r=7 and r=10. The code for r=1 could be quite effective in a binary-coded decimal machine. It is also interesting to observe that, due to irregularities in the existence of good codes with r at least 4, the redundancies of the residue codes are occasionally less than the comparable AN codes. Several such examples exist in Table III.

A source of complexity arises when a truncated code is used, e.g., a code of Theorem 7. The implicit truncation leads to the need for an internal overflow correction, requiring some increase in circuitry.

The systematic multi-residue codes and the gAN codes have advantages over the AN codes due to the visibility of their information digits. The multi-residue codes have the disadvantage that the check digits are not directly protected by the code as they are in the AN and gAN codes. Detailed comparison of these approaches is desirable for byte correction. However, the results here apply to all these types of arithmetic codes. Another approach to error correction of iterative errors resulting from a single fault in high-speed arithmetic is found in [7]. Further discussion of systems aspects are found in [15].

X CONCLUSIONS

The codes presented here have considerable potential in the realization of cost-effective fault-tolerant computing systems capable of high availability. They contribute a new approach to the design of byte-sliced arithmetic processors.

XI ACKNOWLEDGMENT

The authors are indebted to Karl N. Levitt for various helpful suggestions.

k	1-bit bytes M	A	R	G	L
16	5	6	7	8	9
24	5	6	7	8	10
32	6	7	8	8	12
48	6	7	8	12	12
64	7	8	9	12	14

k	2-bit bytes M	A	R	G	L
16	6	8	7	8	9
24	6	8	8	8	10
32	6	8	8	8	12
48	7	8	9	12	14
64	7	8	9	12	14

k	4-bit bytes M	A	R	G	L
16	8	10	9	9	9
24	8	11	10	11	11
32	8	11	11	13	13
48	8	12	12	14	16
64	9	12	12	14	16

Table III.
Redundancy for byte correcting codes:
M = Memory error correcting
A = AN, gAN arithmetic error correcting
R = Multiresidue arithmetic error correcting
G = R with generalized low-cost residues only
L = R with low-cost residues only

APPENDIX: PROOFS OF THE RESULTS

Proof of Theorem 1: We prove (2) using the concept that an AN code (with arithmetic distance at least 3 in radix r) is single-byte error correcting if distinct errors have distinct syndromes.* Consider two distinct byte errors $E_1 = \alpha r^j$ and $E_2 = \beta r^\ell$, with $0 < |\alpha| < r$, $0 < |\beta| < r$, for j and ℓ among $0, 1, \ldots, e_r(p) - 1$, $j \neq \ell$. Suppose their syndromes are equal, which is to be proven impossible. (Assume $j > \ell$, without loss of generality.) Then,

$$\alpha r^j \equiv \beta r^\ell \pmod{A}, \text{ and } \alpha r^{j-\ell} \equiv \beta \pmod{A},$$

i.e.,
$$\alpha r^t \equiv \beta \pmod{A}, \tag{12}$$

for $t = j-\ell$, some integer among $1, \ldots, e_r(p) - 1$. It follows from (12) that for this i

$$\alpha r^i \equiv \beta \pmod{r-1} \tag{13}$$

and
$$\alpha r^i \equiv \beta \pmod{p}. \tag{14}$$

From (13) we have $\alpha \equiv \beta \pmod{r-1}$, since $r^i \equiv r \equiv 1 \pmod{r-1}$. There are two cases. If α and β have the same sign, then they are equal. When they are of opposite sign, choose α to be positive without loss of generality, whence

$$\alpha = \beta + (r-1). \tag{15}$$

When $\alpha = \beta$, (14) cannot be satisfied for any i, $0 < t < e_r(p)$, whence the syndromes of all such distinct errors must be distinct, as is to be proved. When $\alpha \neq \beta$, substitute (15) in (14), whence $\alpha r^i \equiv \alpha-(r-1) \pmod{p}$, and $(\alpha-r+1)\,\alpha^{-1} \equiv r^i \pmod{p}$, which implies that (by definition)

$$(\alpha-r+1)\,\alpha^{-1} \in G_r(p) \quad .$$

This is a contradiction of the hypothesis (1), implying that the two syndromes must be distinct. Finally, if $p-1$ is not in $G_r(p)$, then $r^{e_r(p)} \equiv 1 \pmod{p}$, the smallest positive power of r having this property. Thus the code can include all code words AN up to (but not including) $r^{e_r(p)} - 1$, which has arithmetic weight two. □

Proof of Corollary 3: In Theorem 2, set $r = 2$, whence $A = p$. We observe that the open interval $(0, r-1)$ is empty, and therefore (1) is trivially satisfied. Thus (4) follows from (3) in Theorem 2. □

Proof of Corollary 4: Set $r = 3$ in Theorem 2. In the open interval $(0, r-1)$ there is only one integer and that is $s = 1$. For that case $(s-r+1)s^{-1} = -1$. Recall that -1 does not exist in $G_r(p)$ (because of the nonprimitivity of $r = 3$--see the text preceding Theorem 2). Therefore the condition (1) is satisfied, and (5) follows from (3) in Theorem 2. □

Proof of Theorem 5: Set $r = 4$ in Theorem 2. In the open interval $(0, r-1)$ exist only $a = 1$ and $s = 2$, for which $(s-r+1)s^{-1} = -(3-s)s^{-1}$ is -2 and $(-1)(-\frac{p-1}{2}) = \frac{p-1}{2}$, respectively. Since -1 is not in $G_4(p)$ (because of the nonprimitivity of $r = 4$, as above), and because $G_4(p)$ is identical to $G_2(p)$ in this case (whence -2 but not $+2$ is primitive), -2 cannot be in $G_4(p)$. For the same reason, $(p-1)/2$ cannot be. (If it were, then -1 would be.) Thus condition (1) holds. □

Proof of Theorem 6: The proof of syndrome uniqueness is identical to that of Theorem 1, except that the ranges of i, j and ℓ are up to $e_r(p)/2 - 1$. This is to avoid ambiguity between positively and negatively signed errors, since $r^{e_r(p)/2} \equiv -1 \pmod{p}$. Consequently the code can include all code words AN up to but not including $(r-1)r^{e_r(p)/2} + (r-1)$ for even r, and

$$(\tfrac{r-1}{2})\, r^{e_r(p)/2} + (\tfrac{r-1}{2}) \text{ for odd } r,$$

the smallest nonzero radix r representations of arithmetic weight two divisible by $(r-1)p$. □

Proof of Lemma 8: We observe that since p is (by definition) a prime, d must be a prime (Catalidi-Fermat, e.g., see [23], p.3). Thus the elements of $G_r(p)$ are precisely the first d consecutive powers of 2, since $G_r(p)$ is here identical to $G_2(p)$, since $\gcd(b,d) = 1$. Therefore we need only prove that $s\cdot 2^\ell \equiv s-r+1 \pmod{2^d-1}$ cannot occur. Assume that it can. Then

$$s\cdot 2^\ell \equiv 2^d - 2^b + a \pmod{2^d-1}$$
and
$$s\cdot 2^\ell \equiv 2^b(2^{d-b} - 1) + s \pmod{2^d-1}. \tag{16}$$

We note that on the righthand side of the congruence (16) we have an integer less than $2^d - 1$ whose binary representation has two parts, the higher order part of value $2^d - 2^b$ and the lower order part (b digits) of value a. Also we note the Hamming weight of this integer (the number of ones in the binary representation) must be at least one greater than the Hamming weight of s. On the other hand, the Hamming weight of $s\cdot 2^\ell \pmod{2^d-1}$ is the same as the Hamming weight of s, for the reason that the multiplication by a power of 2 modulo $2^d - 1$ is in effect a cyclic shift of s by ℓ places and the Hamming weight is invariant under cyclic shifts. Therefore the congruence (16) cannot hold. □

Proof of Theorem 9: From Lemma 8, condition (1) of Theorem 1 is satisfied. Further, the elements of $G_r(p)$ are of the form $2^k (k = 0, 1, \ldots d-1)$, and $p-1$ clearly is not in $G_r(p)$. Also, since the gcd of b and d is 1, we have $e_r(2^d-1) = d$. Thus (10) follows from Theorem 1. □

Proof of Theorem 10: If $r > \frac{p}{2^f-1}$, then the two errors $\frac{p}{2^f-1} \cdot r^f$ and $\frac{p}{2^f-1} \cdot r^0$ have identical syndromes, violating error correction. This follows simply from $r^f-1 \equiv 0 \pmod{r-1}$ and $r^f-1 = (2^f)^b-1 \equiv 0 \pmod{2^f-1}$, whence $A = (r-1)p = (r-1)\cdot\frac{p}{2^f-1}\cdot(2^f-1)$, which divides the difference of the errors, $\frac{p}{2^f-1} \cdot (r^f-1)$. If $r \leq \frac{p}{2^f-1}$, then these errors cannot arise as single byte errors. It is readily seen that 2^d-1 cannot divide $s(r^i-1)$ in any other way for $1 \leq i < d$, and thus all single-byte error syndromes are distinct. □

---

*(The error $+r^{e_r(p)}$ is naturally excluded. Massey [13] shows that otherwise, and in general for other than single errors, it is not necessary for all syndromes to be distinct for correction of a given set of errors.)

REFERENCES

1. A. Avizianis, Concurrant Diagnosis of Arithmetic Processors, Digest IEEE 1st Annual Computer Conf., pp. 34-37, Sept 1967. See also referances to earlier work contained therein, notably to 1964-66 JPL reports.

2. A. Avizianis, Digital fault diagnosis by low cost arithmetical coding techniques, Proc. Purdue Centenn. Year Symp. Inform. Proc., vol. 1, Lafayette Ind: Purdue Univ. Eng. Exp. Sta., pp. 81-91, April 28-30, 1969.

3. A. Avizienis et al., The STAR (self-Testing and Repairing) Computer: An investigation of the theory and practice of fault-tolerant computer design, IEEE Trans. Electr. Comp. C-20, pp. 1312-1321, Nov 1971.

4. A. Avizianis, Arithmetic error codes: cost and effectivenass studies for application in digital system design, IEEE Trans.Comp. C-20, pp. 1322-1331, Nov 1971.

5. D.T. Brown, Error detecting and error correcting binary codes for arithmetic operations, IRE Trans. Electr. Comp. EC-9, pp. 333-337, Sept 1960.

6. T.L. Cauthan and T.R.N. Rao, Analog techniques for residue operations, Proc. IEEE-TCCA Symp. on Computer Arithmetic, Univ. Md., College Park Md, May 15-16 1972.

7. R. T. Chien and S. J. Hong, Error correction in high-speed arithmetic, IEEE Trans. Comp. C-21, pp. 433-438, May, 1972.

8. R.E. Forbes et al., A self-diagnosable computer, FJCC, pp. 1073-1086, 1965.

9. H.L. Garner, Error codes for arithmetic operations, IEEE Trans. Electr. Comp. EC-15, pp. 763-770, Oct 1966.

10. J. Goldberg, K. N. Levitt and J. H. Wensley, An organization for a highly survivable memory, Digest 1973 Int. Symp. on Fault-Tolerant Comp., Palo Alto CA, pp. 59-64, June 20-22, 1973.

11. S. J. Hong and A. M. Patel, A general class of maximal codes for computer applications, IEEE Trans. Comp. C-21, pp. 1322-31, Dec. 1972.

12. G.G. Langdon and C.K. Tang, Concurrent error detection for group look-ahead binary adders, IBM J.Res.Dev., Sept 1970.

13. J.L. Massey, Survey of residue coding for arithmetic errors, ICC Bull. vol 3, no. 4, Oct 1964.

14. J.L. Massey and O.N. Garcia, Arithmetic codes, (in Advances in Computing, ed. J. Tou), Planum Publ., 1972.

15. P. G. Naumann, J. Goldberg, K. N. Levitt and J. H. Wensley, A Study of Fault-Tolerant Computing: Final Report, Stanford Research Institute, Menlo Park CA, July 1973.

16. B. Parhami and A. Avizianis, Application of arithmetic error codes for checking of mass memories, Digest 1973 Int. Symp. on Fault-Tolerant Comp., Palo Alto CA, pp. 47-51, June 20-22, 1973.

17. W. W. Peterson and E. J. Weldon, Jr., Error Correcting Codes (2nd. edition)., MIT Press, Cambridge, Mass., 1972. First edition by Peterson, 1961, valid alternative reference.

18. T.R.N. Rao, Error checking logic for arithmetic type operations of a processor, IEEE Trans. Electr. Comp. C-17, ppp. 845-849, Sept 1968.

19. T.R.N. Rao, Birasidue error correcting codes for computer arithmetic, IEEE Trans. Comp. C-19, pp. 398-402, May 1970.

20. T.R.N. Rao, Error correction in adders using systematic subcodes, IEEE Trans.Comp, pp. 254-259, Mar 1972

21. T.R.N.Rao and O.N.Garcia, Cyclic and multiresidue codes for arithmetic operations, IEEE Trans. Info. Theory C-21, pp. 85-91, Jan 1971.

22. T.R.N. Rao and A. K. Trahan, Single error correcting non-binary arithmetic codes, IEEE Trans. Info. Theory IT-16, pp. 604-608, Sept 1970.

23. D. Shanks, Solved and Unsolved Problems in Number Theory, vol. 1, Spartan Books, Washington DC, 1962.

24. J. J. Stiffler, The SERF fault-tolerant computer. Part I: conceptual design, Digest 1973 Int. Symp. on Fault-Tolerant Comp., Palo Alto CA, pp. 23-26, June 20-22, 1973.

BASIC DISTRIBUTION LIST (69 copies, 1 each except as noted)

Director, ARPA, Arlington VA 22209 (3 copies)

Defense Documentation Center, Alexandria VA 22314 (12 copies)

Office of Naval Research Code 430C, Arlington VA 22217 (2)

Office of Naval Research, Boston Office, Boston, Mass. 02210

Office of Naval Research, Chicago Office, Chicago Ill. 60605

Office of Naval Research, Pasadena Office, Pasadena, CA 91101

Director, Naval Research Lab, Library, Code 2029, Wash DC 20390 (6)

Commandant of the Marine Corps, Dr. A. L. Slafkosky, Scientific Advisor, Wash DC 20380

Office of Naval Research, Code 427, Arlington VA 22217

Office of Naval Research, New York Area Office, New York NY 10011

Office of Naval Research, San Francisco Area Office, San Francisco CA 94102

Office of Naval Research, Contract Administrator Southeastern Area, Wash DC 20037

Dr. Paul Richards, Naval Research Laboratory, Code 7800, Wash DC 20390

Dr. Bruce Wald, Naval Research Laboratory, Code 5030, Wash DC 20390

Chief of Naval Operations Dept. of the Navy, Wash DC 20350 (2)

Commander, Naval Material Command, Dept. of the Navy Wash DC 20360 (2)

Commander, Naval Air Systems Command, Dept. of the Navy, Wash DC 20360 (2)

Mr. Ron Entner, Naval Air Systems Command, Dept. of the Navy, Wash DC 20360

Commander, Naval Electronics Systems Command, Dept. of the Navy, Wash DC 20360 (2)

Commander, Naval Ordnance Systems Command, Dept. of the Navy, Wash DC 20360 (2)

Commander, Naval Ship Systems Command, Dept. of the Navy, Wash DC 20360 (2)

Commander, Naval Ship Engineering Center, Hyattsville MD 20782 (2)

Mr. Gene H. Gleissner, Naval Ship Research and Development Center, Dept. of Applied Mathematics, Bethesda MD 20034

U. S. Naval Ordnance Laboratory, White Oak, Silver Spring MD 20910

Naval Weapons Center, China Lake CA

Commander, Naval Weapons Laboratory, Dahlgren VA 22448 (2)

Dr. John B. Slaughter, Naval Electronics Laboratory Center, San Diego CA 92152

Commanding Officer, Fleet Computer Programming Center, Pacific, San Diego CA 92147

Commander, Naval Air Development Center, Code AEDC, Warminster PA 18974

Commanding Officer, Naval Underwater Sound Laboratory, New London Conn. 06320

Dr. Lawrence Roberts, 730A8 - ARPA, Arlington VA 22209

Dr. Robert E. Kahn, 730A8 - ARPA, Arlington VA 22209

Mr. Stephen Crocker, 730Ab - ARPA, Arlington VA 22209

Dr. I. R. Herschman, Office of the Chief, Research and Development, U. S. Army, Arlington VA 20315

Mr. M. Andrews, Air Force Office of Scientific Research, Arlington VA 22209

Lt. Col. T. J. Wachowski, Air Force Office of Scientific Research, Arlington VA 22209

National Bureau of Standards, Wash DC 20234

Mr. J. Lehman, National Science Foundation, Wash DC 20550

Mr. N. Murray, NASA-Langley, Hampton VA 23365

Mr. R. Nelson, RADC, Rome NY 13441

Mr. J. R. Suttle, USARO, Durham NC 27706

Mr. O. Brewer, AFAL/AAM Wright-Patterson Air Force Base, Ohio 45433

SELECTED ONR CONTRACTORS (9 copies, 1 copy each)

Dr. O. Cesaseni, Carnegie-Mellon Univ., Dept. of Electrical Engineering, Pittsburgh PA 15213

Dr. W. Chu, Univ. of California, L. A., Dept of Computer Science, Los Angeles CA 90024

Mr. Roy Hunter, John Hopkins University Appl. Physics Lab. 8621 Georgia Ave, Silver Spring, MD 20910

Dr. H. Knoebel, Univ. of Illinois, Coordinated Science Laborstory, Urbana Ill 61801

Dr. W. J. Poppelbaum, Univ. of Illinois, Digital Computer Laboratory, Urbana Ill 61801

Prof. D. A. Rudberg, Montana State Univ., Electrical Engineering Dept., Bozeman Mont. 59715

Mr. Robert Coleman, Naval Weapons Center, Code 3031, China Lake CA 93555

Prof. C. M. Allen, State University of New York at Buffalo, Dept. of Electrical Engineering, Buffalo NY 14214

Dr. M. A. Breuer, Univ. of Southern California, Electronic Sciences Laboratory, Los Angeles, CA 90007

ARPA CONTRACTORS (37, 1 copy each)

Mr. Robert P. Abbott, Lawrence Livermore Lebore ory, P. O. Box 808, Livermore CA 94550

Professor Jonathan Allen, MIT, Cambridge, Mass. 02139

Dr. Roy Amara, Institute for the Future, Menlo Park, CA 94024

Dr. Robert Balzer, Univ. Southern California, Information Sciences Institute, Marina Del Ray, CA 90291

Mr. Paul Baran, Cobledote Associates, Inc., Menlo Park, CA 94025

Professor Herbert B. Baskin, University of California, Berkeley, CA 94720

Mr. Morton L. Bernstein, System Development Corporation, Sante Monica, CA 90406

Mr. Roland F. Bryan, Computer Systems Lab, University of California, Sente Barbara, CA 93106

Professor Thomas E. Cheetham, Jr., Harverd Univereity, Aiken Computetion Lab, Cambridge, Mass. 02138

Dr. Douglas C. Engelbert, SRI, Menlo Park, CA 94025

Professor David C. Evans, University of Utah, Salt Lake City Utah 84112

Professor Edward A. Feigenbaum, Computer Science Department, Stanford University, Stanford, CA 94305

Mr. James W. Forgie, MIT, Lincoln Laboretory, Lexington, Mass. 02173

Dr. Howard Frank, Network Analysis Corporation, Glen Cove, NY 11542

Professor Edward Fredkin, MIT, Project MAC, 545 Technology Square, Cambridge, Mass. 02139

Professor E. L. Glaser, Case Western Reserve Univereity, Clevelend, Ohio 44106

Dr. Peter Hart, SRI, Menlo Park, CA 94025

Mr. Frank Heart, Bolt, Berenek & Newman, Cambridge, Mass 02138

Dr. Anetol W. Holt, Massachusette Computer Associates, Wakafield, Mass. 01880

Professor Leonard Kleinrock, Computer Science Department, UCLA, Los Angeles CA 90024

Professor Franklin Kuo, Univereity of Hawaii, Honolulu, Hawaii 96822

Professor J. C. R. Licklider, MIT, Project MAC, 545 Technology Square, Cambridge, Mass. 02139

Dr. Thomas Marill, Computer Corporation of America, Cambridge, Mass. 02139

Professor John McCarthy, Artificial Intelligence Laboratory, Stanford University, Stanford, CA 94305

Professor Marvin Minsky, Artificial Intelligence Laboratory, MIT, 545 Technology Square, Cambridge, Mass. 02139

Dr. James G. Mitchell, Xerox PARC, Computer Science Laboratory, Palo Alto CA 94304

Professor Cherles E. Molnar, Computer Syatems Laboratory, Washington Univeraity, St. Louis, MO 63110

Professor Allen Newell, Computer Sceinca, Carnegie-Mellon Univeraity, Pittsburgh, PA 15213

Dr. Mel Pirtle, NASA/Ames Research Canter, Moffett Field, CA 94035

Professor William K. Pratt, USC, Los Angeles, CA 90007

Professor J. A. Robinson, Syracuse Univeraity, Systems and Information Sciences, Syracuse, N.Y. 13210

Professor John Sevega, Brown Univeraity, Division of Engineering, Providence, RI 02912

Dr. June E. Shoup, Speech Communicetions Research Laboratory, Sente Barbara CA 93109

Professor Daniel L. Slotnick, Center for Advanced Computation, Univeraity of Illinois, Urbana, Ill. 61801

Professor Thomas G. Stockham, Jr., Computer Science, University of Utah, Salt Lake City, Utah 84112

Dr. William R. Sutherland, Bold, Beranek & Newman, Cambridge, Mass. 02138

Mr. Keith W. Uncapher, Univ. Southern California, Information Sciences Institute, Marine Del Ray, CA 90291

CONTRIBUTORS TO APPENDIX 2 (16 copies, 1 copy each)

Prof. A. Avizienis, JPL and UCLA

Mr. B. R. Borgerson, U. C. Berkeley, CA

Dr. W. C. Carter, IBM, Yorktown Haights, NY

J. L. Delemere, EMD, St.-Cloud, France

Capt. L. A. Fry, SAMSO, Los Angeles, CA

Prof. A. L. Hopkins, Jr., MIT Draper Lab, Cambridge MA

Mr. L. J. Koczela, North-American Rockwell, Anaheim CA

Mr. W. L. Martin, Hughes Aircraft, Fullerton CA

Mr. J. S. Miller, Intermatrics, Cambridge MA

Mr. S. M. Ornstein, Bolt Beranek & Newman, Cambridge MA

Dr. W. C. Ridgway III, Bell Labs, Madison NJ

Prof. J. H. Saltzer, MIT Project MAC, Cambridge MA

Prof. D. Siewiorek, Carnegie-Mellon Univ, Pittsburgh PA

Dr. W. Ulrich, Bell Labs, Naperville, Ill.

Pacific Coast Stock Exchange, San Fran/Los Angeles, CA

R. K. Williams, Plessey, Teplow, England